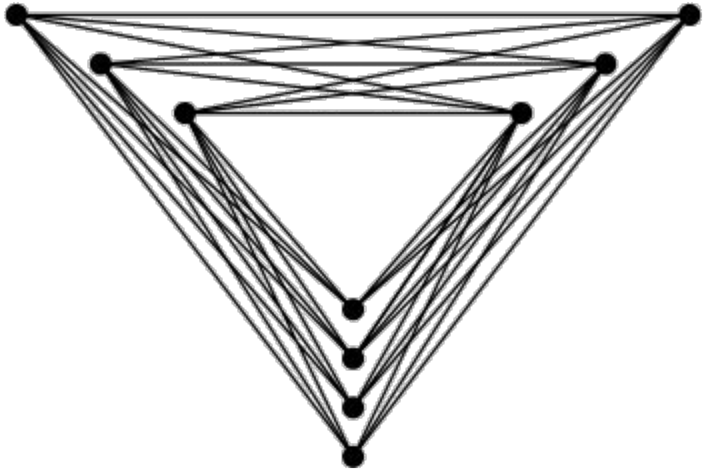
Departamento de Ingeniería Informática
Universidad de Santiago de Chile

Teoría de la Computación

Primer semestre 2024

Daniel Vega Araya



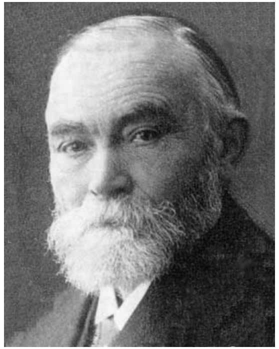
Santiago, 2024

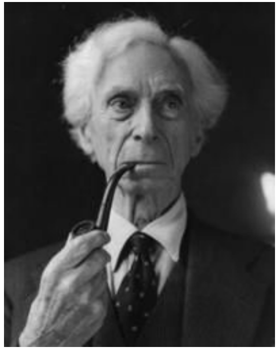
demostración*...

- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.

demostración*...

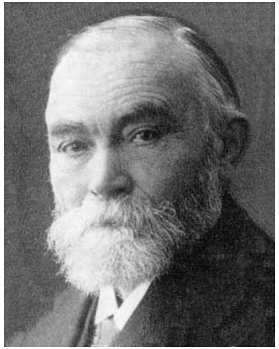
- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.
    - Frege y Russell intentan reducir las matemáticas a la lógica y la teoría de conjuntos.



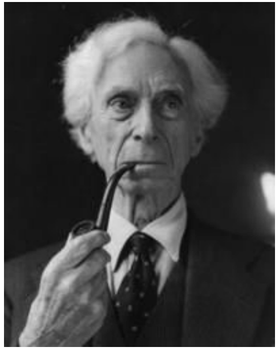


demostración*...

- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.
    - Frege y Russell intentan reducir las matemáticas a la lógica y la teoría de conjuntos.



PARADOJA



demostración*...

En un lejano poblado de un antiguo emirato había un barbero llamado As-Samet *diestro en afeitar cabezas y barbas, maestro en escamondar pies y en poner sanguijuelas*. Un día el emir se dio cuenta de la falta de barberos en el emirato, y ordenó que los barberos sólo afeitaran a aquellas personas que no pudieran afeitarse. Y así mismo impuso la norma de que todo el mundo se afeitase, (no se sabe si por higiene, por estética, o por demostrar que podía imponer su santa voluntad y mostrar así su poder). Cierto día el emir llamó a As-Samet para que lo afeitara y él le contó sus angustias:

—En mi pueblo soy el único barbero. No puedo afeitar al barbero de mi pueblo, ¡que soy yo!, ya que si lo hago, entonces puedo afeitarme por mí mismo, por lo tanto ¡no debería afeitarme! pues desobedecería vuestra orden. Pero, si por el contrario no me afeito, entonces algún barbero debería afeitarme, ¡pero como yo soy el único barbero de allí!, no puedo hacerlo y también así desobedecería a vos mi señor, oh emir de los creyentes, ¡que Allah os tenga en su gloria!

El emir pensó que sus pensamientos eran tan profundos, que lo premió con la mano de la más virtuosa de sus hijas. Así, el barbero As-Samet vivió para siempre feliz y barbón.[2]

demostración*...

- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.
  - Frege y Russell intentan reducir las matemáticas a la lógica y la teoría de conjuntos.

demostración*...

- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.
  - Frege y Russell intentan reducir las matemáticas a la lógica y la teoría de conjuntos.
- Otro intento fue el de David Hilbert (1862-1943), iniciador de la teoría de la demostración o metamatemática.
  - Buscaba encontrar un modo de demostrar la **consistencia** de cualquier lista de axiomas.

demostración*...

- Múltiples tentativas a comienzos del siglo XX de fundamentar la matemática sobre sólidas bases lógicas.
  - Frege y Russell intentan reducir las matemáticas a la lógica y la teoría de conjuntos.
- Otro intento fue el de David Hilbert (1862-1943), iniciador de la teoría de la demostración o metamatemática.
  - Buscaba encontrar un modo de demostrar la **consistencia** de cualquier lista de axiomas.
- Hilbert tenía una concepción de las matemáticas que denominaba formalismo.
  - Las cosas de las que hablan las matemáticas no son más que símbolos.
  - Los símbolos carecen de significado por sí mismos: Lo sabemos todo sobre ellos cuando comenzamos a manipularlos.
  - Estableció reglas recursivas para explicar sus posibles interacciones.

demostración...

- Hilbert planteó la búsqueda de un procedimiento algorítmico general válido para resolver todas las posibles cuestiones matemáticas.
- Su planteamiento buscaba obtener la respuesta a tres importantes preguntas:
  - **¿Son las matemáticas completas?** (cualquier proposición puede ser probada o rechazada)
  - **¿Son las matemáticas consistentes?** (no se producen contradicciones)
  - **¿Son las matemáticas decidibles?** (cualquier proposición se puede demostrar tras una secuencia finita de pasos)

demostración...

- Hilbert planteó la búsqueda de un procedimiento algorítmico general válido para resolver todas las posibles cuestiones matemáticas.
- Su planteamiento buscaba obtener la respuesta a tres importantes preguntas:
  - **¿Son las matemáticas completas?** (cualquier proposición puede ser probada o rechazada)
  - **¿Son las matemáticas consistentes?** (no se producen contradicciones)
  - **¿Son las matemáticas decidibles?** (cualquier proposición se puede demostrar tras una secuencia finita de pasos)

completitud...

- Los planteamientos de Hilbert atrajeron una multitud de investigadores, entre ellos Kurt Gödel (1906 – 1978).

El problema de la completitud consiste en demostrar que **un sistema axiomático**, del tipo de los que Hilbert propuso, es capaz de demostrar o derivar toda proposición verdadera dentro del sistema.

- En 1929 demostró la completitud del cálculo de predicados de primer orden o lógica de primer orden.

Ninguna proposición verdadera podía escapar del poder demostrativo de este tipo de lógica.

demostración...

- Hilbert planteó la búsqueda de un procedimiento algorítmico general válido para resolver todas las posibles cuestiones matemáticas.
- Su planteamiento buscaba obtener la respuesta a tres importantes preguntas:
  - **¿Son las matemáticas completas?** (cualquier proposición puede ser probada o rechazada)
  - **¿Son las matemáticas consistentes?** (no se producen contradicciones)
  - **¿Son las matemáticas decidibles?** (cualquier proposición se puede demostrar tras una secuencia finita de pasos)

Incompletitud

- La **incompletitud** implicaba la falta de axiomas, pues se suponía que una proposición verdadera no demostrable podría serlo agregando nuevos axiomas.
- En 1931 demostró que cualquier sistema axiomático computable que sea capaz de describir la aritmética de los números naturales (por ej. los Axiomas de Peano), está sujeto a las siguientes condiciones:
  - Si el sistema es consistente entonces no puede ser completo, y
  - La consistencia de los axiomas no puede demostrarse desde dentro del sistema.

Incompletitud

- La **incompletitud** implicaba la falta de axiomas, pues se suponía que una proposición verdadera no demostrable podría serlo agregando nuevos axiomas.

- En 1931 demostró que cualquier sistema axiomático **computable** que sea capaz de describir la aritmética de los números naturales (por ej. los Axiomas de Peano), está sujeto a las siguientes condiciones:
  - Si el sistema es consistente entonces no puede ser completo, y
  - La consistencia de los axiomas no puede demostrarse desde dentro del sistema.

Hilbert, we have a problem!

demostración...

- Hilbert planteó la búsqueda de un procedimiento algorítmico general válido para resolver todas las posibles cuestiones matemáticas.
- Su planteamiento buscaba obtener la respuesta a tres importantes preguntas:
  - ¿Son las matemáticas completas?, es decir cualquier proposición puede ser probada o rechazada
  - ¿Son las matemáticas consistentes?, es decir no es posible demostrar algo falso
  - **¿Son las matemáticas decidibles?, es decir cualquier proposición se puede demostrar como cierta o falsa tras una secuencia finita de pasos**

Computabilidad... Entscheidungsproblem

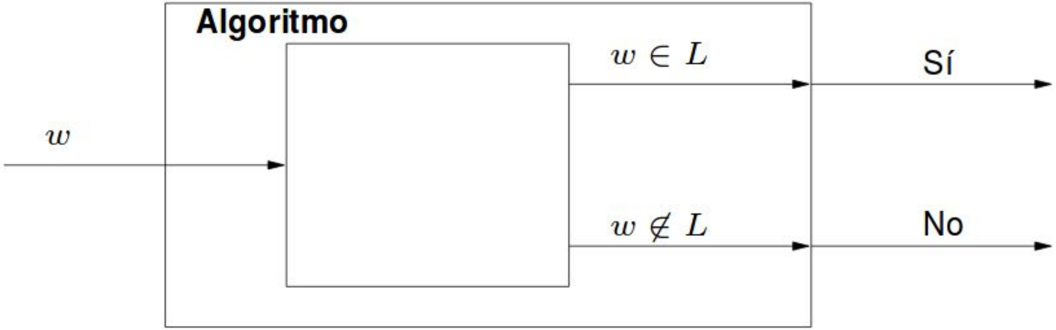
- Entscheidungsproblem o **problema de decisión**
- Definir formalmente la noción de algoritmo.
- **Alonzo Church** en 1936: concepto de "calculabilidad efectiva" basada en el cálculo lambda.
- **Alan Turing** basándose en la máquina de Turing.
- Los dos enfoques son equivalentes: pueden resolver exactamente los mismos problemas.

Computabilidad... Entscheidungsproblem

Dado un problema de decisión cualquiera P, con un lenguaje asociado L, diremos que P y L son decidibles, si es posible encontrar un algoritmo tal que, dada cualquier entrada w pueda responder SÍ, si $w \in L$ y NO si $w \notin L$.

Computabilidad... Entscheidungsproblem

Dado un problema de decisión cualquiera P, con un lenguaje asociado L, diremos que P y L son decidibles, si es posible encontrar un algoritmo tal que, dada cualquier entrada w pueda responder SÍ, si $w \in L$ y NO si $w \notin L$.

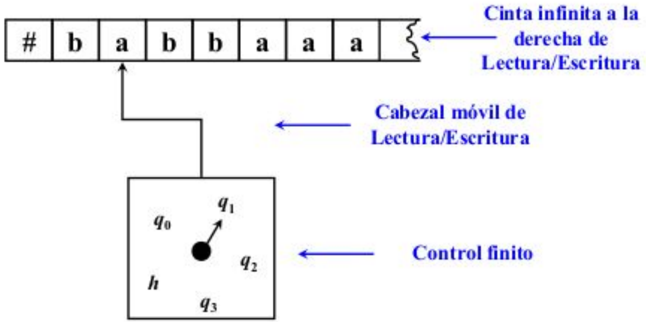


Máquinas de Turing

- Introducida por el matemático-lógico inglés Alan Turing (1912 – 1954) alrededor de la segunda guerra mundial antes de que existieran los lenguajes de programación.
- Son más generales que las máquinas anteriores.
- Representan una clase maximal y estable de autómatas.
- Fueron diseñadas satisfaciendo tres criterios fundamentales de forma simultánea:
  - MT deben ser autómatas. Siguen el espíritu de los autómatas anteriores.
  - Deben ser lo más simples de describir (definiciones formales y razón de ser).
  - Deben ser lo más general en términos de computaciones que ellos puedan realizar.

Máquinas de Turing

- Definición: Una Máquina de Turing es una tupla *(Q, Σ, δ, s, F)*
  - *Q* es un conjunto finito de estados
  - *Σ* es el alfabeto: conjunto finito de símbolos de entrada.
  - *δ* función de transición
    - $\delta: Q \times \Sigma \quad Q \times \Sigma \; U \; \{I, D\}$
  - *s* representa el símbolo inicial (s pertenece a Q).
  - *F* es un conjunto de estados finales (no vacío).
  - se distingue el estado final q_h



Máquinas de Turing - **Funcionamiento**

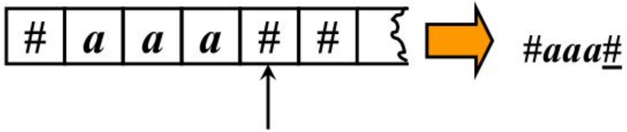
- **Cabezal**: lee y escribe símbolos del alfabeto en la cinta
- **Control finito**: opera en pasos discretos. En cada paso puede realizar:
  - Pasar a un nuevo estado
  - Tanto:
    - escribir/leer un símbolo en la posición que se encuentra el cabezal en la cinta, o…
    - ...mover el cabezal un espacio a la izquierda o a la derecha.
- **Cinta**: tiene inicio a la izquierda, pero es infinita a la derecha. Puede contener símbolos ‘*blancos*’, simbolizados como ‘#’.

Por convención, el cabezal comienza en el ‘#’ que continúa al último símbolo de la palabra en la cinta, i.e., #abc<u>#</u>

Máquinas de Turing - **Funcionamiento**

- Considere la máquina $M = (Q, \Sigma, \delta, s, F)$
  - $Q=\{q_0 , q_1\}$
  - $\Sigma=\{a\}$
  - $s=q_0$
  - $F=\{q_h\}$
  - $\delta=$

q	σ	δ(q, σ)
q_0	a	*
q_0	#	(q_1,I)
q_1	a	$(q_0,\#)$
q_1	#	(q_h,D)



Máquinas de Turing - **Funcionamiento**

- Construir una máquina que cambia 1 por 0 (complemento binario)
  - $Q=\{q_0, q_1\}$
  - $\Sigma=\{0,1\}$
  - $s=q_0$
  - $F = \{q_1\}$
  - $\delta=$

MT - **Computación**

Definición: Una computación de una máquina de Turing *M* es una secuencia de configuraciones $C_0, C_1, \ldots, C_n$ para algún *n* >1.

$$C_0 \vdash_M C_1 \vdash_M \cdots \vdash_M C_n$$

MT - **Computación**

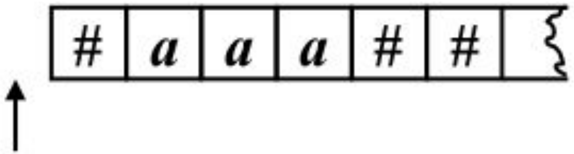
Definición: Una computación de una máquina de Turing *M* es una secuencia de configuraciones $C_0, C_1, ..., C_n$ para algún *n* >1.

$$C_0 \vdash_M C_1 \vdash_M \dots \vdash_M C_n$$

y si se configura mal la MT?

MT - **Configuración de colgado**

Una configuración de colgado (*hanging configuration*) se presenta cuando la máquina **no para su ejecución** o **se encuentra una posición a la izquierda del principio de la cinta**.



MT - **Computan**

Computar es calcular.

MT - **Computan**

Computar es calcular. En términos generales, un sistema computa cuando es capaz de captar, almacenar y representar información, para generar un resultado a partir de un conjunto determinado de pasos o reglas.

MT - **Computan**

- Las MT computan funciones de strings a strings.
- Recibe como entrada un string w delimitado por blancos.
- Devuelve en la cinta el valor de *f(w)*.

MT - **Computan**

- Las MT computan funciones de strings a strings.
- Recibe como entrada un string w delimitado por blancos.
- Devuelve en la cinta el valor de *f(w)*.

Si existe una MT *M* que compute *f*, se dice que la función *f* es **Turing-Computable**.

MT - **Computan**

- Las MT computan funciones de strings a strings.
- Recibe como entrada un string w delimitado por blancos.
- Devuelve en la cinta el valor de *f(w)*.

Si existe una MT *M* que compute *f*, se dice que la función *f* es **Turing-Computable**.

- Se pueden extender estos conceptos a funciones que reciban cero o más argumentos… incluso a funciones de IN a IN.

MT - **Computan**

- Las MT computan funciones de strings a strings.
- Recibe como entrada un string w delimitado por blancos.
- Devuelve en la cinta el valor de *f(w)*.

Si existe una MT *M* que compute *f*, se dice que la función *f* es **Turing-Computable.**

- Se pueden extender estos conceptos a funciones que reciban cero o más argumentos… incluso a funciones de IN a IN.
  - **Se debe definir la manera en que se codificará la información.**

MT - **Computan**

Ejemplo:

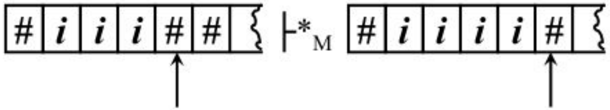
Sea *f* la función del sucesor: $f(n) = n + 1$, para cada n en IN

MT - **Computan**

Ejemplo:

Sea *f* la función del sucesor: *f(n)* = n + 1, para cada n en IN

La MT *M* que realiza esta función es la que sigue:



M = (Q, Σ, δ, s)

Q = $\{q_0\}$

Σ = {i}

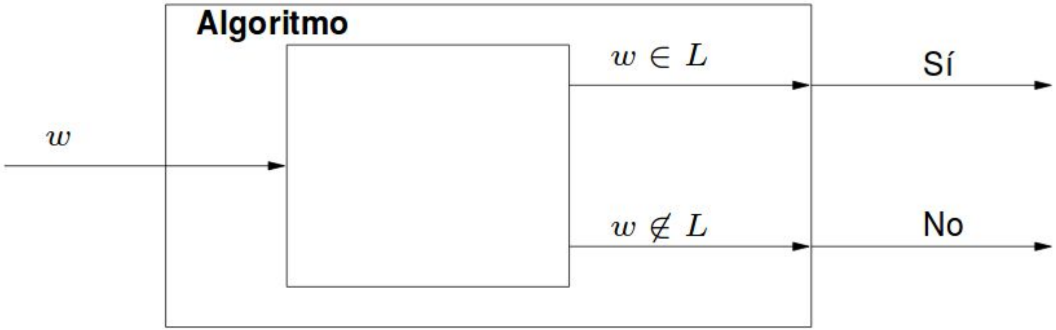
s = q_0

δ=

q	**σ**	**δ(q, σ)**
q_0	*i*	*(h, D)*
q_0	#	$(q_0$,i)

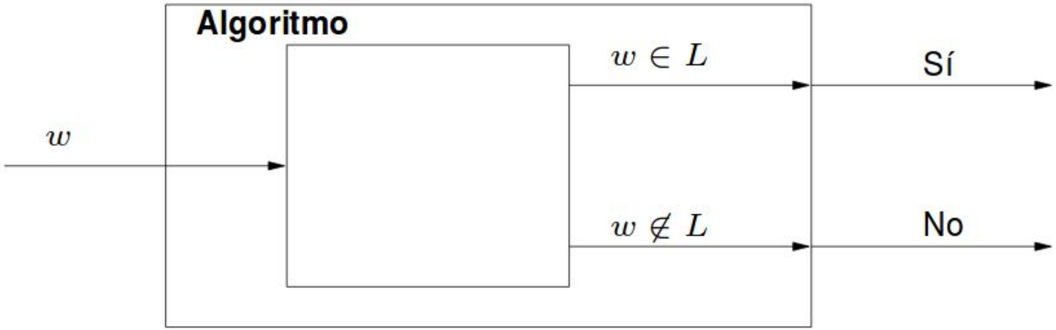
MT - **Problemas de decisión**

- Problema de decisión asociado a un lenguaje L: Dado w perteneciente a Σ*, decidir si w pertenece a L.
- Por ejemplo:
  - Σ = {0,1,2,3,4,5,6,7,8,9}
  - Un número n cualquiera (formado a partir de Σ), ¿es primo?
    (Sólo dos alternativas: SÍ o NO)



MT - **Problemas de decisión**

- Problema de decisión asociado a un lenguaje L: Dado w perteneciente a Σ^*, decidir si w pertenece a L.
- Por ejemplo:
  - $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
  - Un número n cualquiera (formado a partir de Σ), ¿es primo?
    (Sólo dos alternativas: SÍ o NO)



veamos...

MT - **Problemas de decisión**

Primeros 10 números primos

MT - **Problemas de decisión**

Primeros 10 números primos

2	3	5	7	11	13	17	19	23	29

MT - **Problemas de decisión**

Primeros 100 números primos

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73
79	83	89	97	101	103	107	109	113	127	131	137	139	149	151	157	163	167	173	179	181
191	193	197	199	211	223	227	229	233	239	241	251	257	263	269	271	277	281	283	293	307
311	313	317	331	337	347	349	353	359	367	373	379	383	389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479	487	491	499	503	509	521	523	541	547	557	563	569	571
577	587	593	599	601	607	613	617	619	631	641	643	647	653	659	661	673	677	683	691	701
709	719	727	733	739	743	751	757	761	769	773	787	797	809	811	821	823	827	829	839	853
857	859	863	877	881	883	887	907	911	919	929	937	941	947	953	967	971	977	983	991	997

MT - **Problemas de decisión**

- Si existe una respuesta (SÍ o NO) para cada w de Σ^*, diremos que el problema es **decidible**.

- En caso contrario diremos que el problema es **indecidible**

MT - **Problemas de decisión**

- Un problema de decisión es aquél en donde las respuestas posibles son SÍ o NO.
- Un lenguaje L cuyas palabras no contienen símbolos ‘#’ es **Turing-Decidible** si y sólo si la función

$$\chi_L(w) = \begin{cases} \checkmark \text{ si } w \text{ está en } L \\ \times \text{ si } w \text{ no está en } L \end{cases}$$

tal que √ y × no son símbolos del lenguaje.

MT - **Problemas de decisión**

- Un problema de decisión es aquél en donde las respuestas posibles son SÍ o NO.
- Un lenguaje L cuyas palabras no contienen símbolos '#' es **Turing-Decidible** si y sólo si la función

$$\chi_L(w) = \begin{cases} \surd \text{ si w está en L} \\ \times \text{ si w no está en L} \end{cases}$$

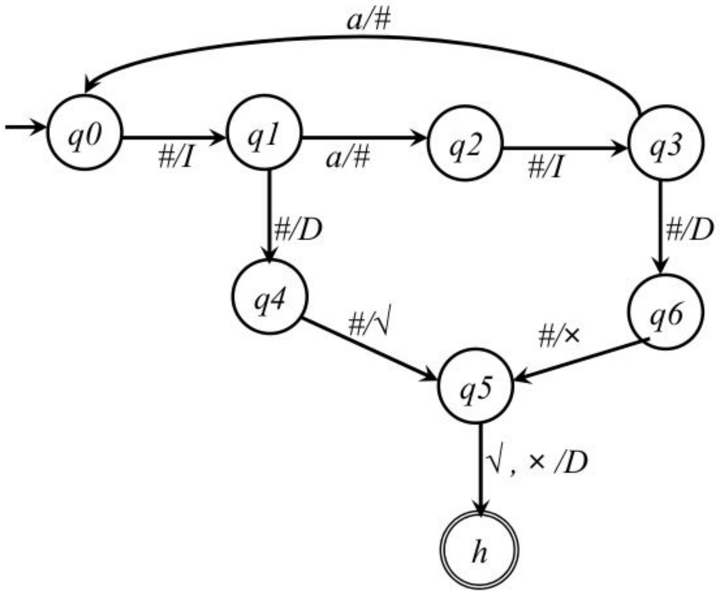
tal que √ y × no son símbolos del lenguaje.

Si existe una MT *M* que computa χ_L(w), para todas las palabras w de L, diremos que ***M* decide el lenguaje L**.

MT - **Problemas de decisión**

Ejemplo:

Dado L = {w pertenece a Σ_0^* / |w| es par}



$\#aaaa\underline{\#} \vdash^*_M \#\surd\underline{\#}$

$\#aaaaa\underline{\#} \vdash^*_M \#\times\underline{\#}$

MT - **Problemas de decisión**

- Un problema de decisión es aquél en donde las respuestas posibles son SÍ o NO.
- Un lenguaje L cuyas palabras no contienen símbolos ‘#’ es **Turing-Decidible** si y sólo si la función

$$\chi_L(w) = \begin{cases} \surd \text{ si w está en L} \\ \times \text{ si w no está en L} \end{cases}$$

tal que √ y × no son símbolos del lenguaje.

- Si existe una MT *M* que computa $\chi_L(w)$, para todas las palabras w de L, diremos que ***M* decide el lenguaje L**.

MT - **Problemas de decisión**

entonces, si

$$\chi_L(w) = \begin{cases} \checkmark \text{ si w está en L} \\ \times \text{ si w no está en L} \end{cases}$$

estamos....

MT - **Problemas de decisión**

entonces, si

$$\chi_L(w) = \begin{cases} \checkmark \text{ si w está en L} \\ \times \text{ si w no está en L} \end{cases}$$

estamos....

ACEPTANDO

MT - **Problemas de decisión**

entonces, si

$$\chi_L(w) = \begin{cases} \checkmark & \text{si w está en L} \\ \times & \text{si w no está en L} \end{cases}$$

estamos….

ACEPTANDO

y aceptar es...

Aceptar

MT - **Aceptan**

- Entenderemos que una MT *M* **acepta** un lenguaje L si ésta **se detiene** **para todas** las palabras que pertenecen al lenguaje L.

- Entenderemos que una MT *M* **no acepta** un lenguaje L si ésta **NO** **se detiene** para todas las palabras que pertenecen al lenguaje L (queda computando por siempre).

MT - **Aceptan**

- Entenderemos que una MT *M* **acepta** un lenguaje L si ésta **se detiene para todas** las palabras que pertenecen al lenguaje L.

- Entenderemos que una MT *M* **no acepta** un lenguaje L si ésta **NO se detiene** para todas las palabras que pertenecen al lenguaje L (queda computando por siempre).

luego, podemos decir que,

- Las MT también son aceptadores de lenguajes.

- Un lenguaje L es **Turing-Aceptable** si y sólo si existe una MT que lo acepte.

MT - **Aceptan**

entonces??...

MT - **Aceptan**

lenguaje **Turing-Decidible**

lenguaje **Turing-Aceptable**

MT - **Aceptan**

lenguaje **Turing-Decidible** ⇄ lenguaje **Turing-Aceptable**

MT - **Aceptan**

lenguaje **Turing-Decidible** $\substack{\Longrightarrow \\ \not\Longleftarrow}$ lenguaje **Turing-Aceptable**

MT - **Aceptan**

lenguaje **Turing-Decidible** $\overset{\Longrightarrow}{\not\Longleftarrow}$ lenguaje **Turing-Aceptable**

¿w está en L**?**

MT - **Aceptan**

lenguaje **Turing-Decidible** $\Rightarrow$ / $\not\Leftarrow$ lenguaje **Turing-Aceptable**

¿w está en L**?**

SI/NO

MT - **Aceptan**

lenguaje **Turing-Decidible** $\Rightarrow$ lenguaje **Turing-Aceptable**

lenguaje **Turing-Decidible** $\not\Leftarrow$ lenguaje **Turing-Aceptable**

¿w está en L?

SI/NO

se detiene/no se detiene

w está en L

MT - **Aceptan**

lenguaje **Turing-Decidible** $\rightarrow$ / $\not\leftarrow$ lenguaje **Turing-Aceptable**

¿w está en L?

SI/NO

se detiene/no se detiene

w **está** en L

MT - **Aceptan**

Ejemplo:

L={w pertenece a Σ_0^* / w contiene al menos una a}

La MT que lo acepta es:

M = (Q, Σ, δ, s) donde
$Q = \{q_0\}$
Σ = {a, b}
$s = q_0$
δ=

q	σ	$\delta(q, \sigma)$
q_0	a	(h, a)
q_0	b	(q_0, I)
q_0	#	(q_0, I)

MT - **Aceptan**

Ejemplo:

L={w pertenece a Σ_0* / w contiene al menos una a}

La MT que lo acepta es:

M = (Q, Σ, δ, s) donde
Q = {q_0}
Σ = {a, b}
s = q_0
δ=

q	σ	$\delta(q, \sigma)$
q_0	a	(h, a)
q_0	b	(q_0, I)
q_0	#	(q_0, I)

Se cuelga si no encuentra una *a*

MT - **otra idea?**

- Los ejemplos mostrados son sumamente simples
- No las debemos subestimar
- Hagamos algo más

MT - **otra idea?**

- Los ejemplos mostrados son sumamente simples
- No las debemos subestimar
- Hagamos algo más:
  - Poseemos máquinas simples

MT - **otra idea?**

- Los ejemplos mostrados son sumamente simples
- No las debemos subestimar
- Hagamos algo más:
  - Poseemos máquinas simples
  - Tenemos imaginación

MT - **otra idea?**

- Los ejemplos mostrados son sumamente simples
- No las debemos subestimar
- Hagamos algo más:
  - Poseemos máquinas simples
  - Tenemos imaginación
  - principal argumento: "Dividir para conquistar".

MT - **otra idea?**

- Los ejemplos mostrados son sumamente simples
- No las debemos subestimar
- Hagamos algo más:
  - Poseemos máquinas simples
  - Tenemos imaginación
  - principal argumento: "Dividir para conquistar".

Las MT se pueden combinar para construir MT grandes y complejas

MT - Combinando... **idea**

- Cada MT simple puede ser considerada como una subrutina o un módulo.

MT - Combinando... **idea**

- Cada MT simple puede ser considerada como una subrutina o un módulo.

- Podemos encadenar **MT básicas** para lograr nuestros objetivos:
  - $M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow \ldots \rightarrow M_k$
  - M_1 prepara la cinta para M2; M2 prepara la cinta para M3 y así sucesivamente.
  - Cada M_i es independiente de las demás.

MT - Combinando... **elementos necesarios**

- Las máquinas básicas son de dos tipos:
  - las que escriben símbolos y no se desplazan
  - las que sólo se mueven por la cinta

MT - Combinando... **elementos necesarios**

- Las máquinas básicas son de dos tipos:
  - las que **escriben símbolos** y no se desplazan
  - las que sólo se mueven por la cinta
- Hay $|\Sigma|$ Máquinas de Turing que **escriben símbolos** de Σ.
  - Formalmente se denota como W_α a la máquina que sólo escribe el símbolo α en la cinta y después para (**la denotaremos como α**).

MT - Combinando... **elementos necesarios**

- Las máquinas básicas son de dos tipos:
  - las que escriben símbolos y no se desplazan
  - las que sólo **se mueven** por la cinta
- Hay |Σ| Máquinas de Turing que escriben símbolos de Σ.
  - Formalmente se denota como W_α a la máquina que sólo escribe el símbolo α en la cinta y después para (**la denotaremos como α**).
- Las MT's **de movimiento** solo mueven la cinta un espacio a la izquierda o a la derecha y luego se detienen.
  - Formalmente se denotan como V_I y V_D a la máquina que se desplaza una posición a la izquierda o a la derecha, respectivamente (**la denotaremos como I o D**).

MT - Combinando... **preparación**

- Las reglas para la combinación de MT básicas:
  - Considerar a cada máquina básica como un estado individual de un autómata.
  - Definir transiciones entre las máquinas básicas igual como se hace en un autómata.
  - La transición de M_1 a M_2 no es gatillada hasta que M_1 detenga su ejecución. La ejecución de M_2 estará en función del estado de la cinta dejado por M_1.

MT - Combinando... **preparación**

- Las reglas para la combinación de MT básicas:
    - Considerar a cada máquina básica como un estado individual de un autómata.
    - Definir transiciones entre las máquinas básicas igual como se hace en un autómata.
    - La transición de M_1 a M_2 no es gatillada hasta que M_1 detenga su ejecución. La ejecución de M_2 estará en función del estado de la cinta dejado por M_1.

$$> M_1 \longrightarrow M_2$$

MT - Combinando... **preparación**

- Las transiciones pueden llevar símbolos en caso de que necesitemos una “condición” para ejecutar otra máquina.

MT - Combinando... **preparación**

- Las transiciones pueden llevar símbolos en caso de que necesitemos una “condición” para ejecutar otra máquina.

$$
\begin{tikzcd}
> M_1 \arrow[r, "a"] \arrow[d, "b"] & M_2 \\
M_3 &
\end{tikzcd}
$$

MT - Combinando... **máquinas básicas**

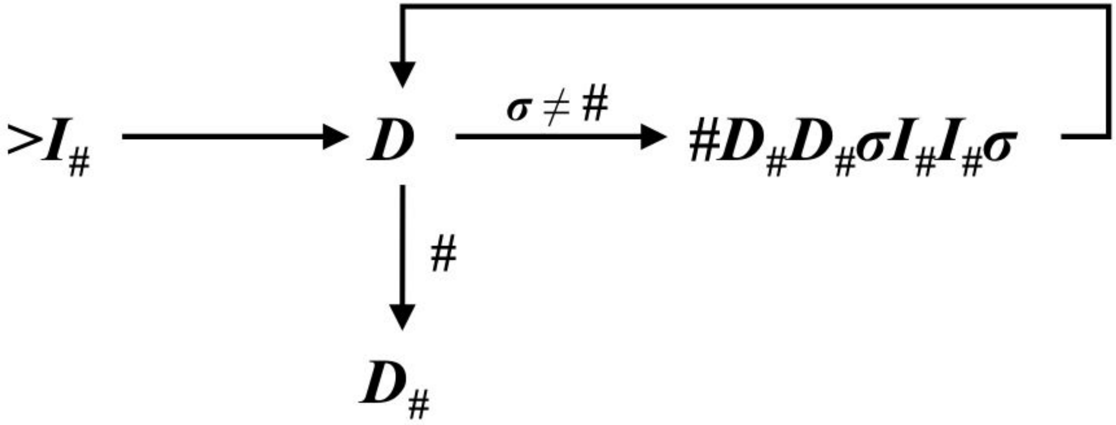
Nosotros utilizaremos,

- $D_{\#}$: se mueve a la derecha y para cuando encuentra un '#'
- $I_{\#}$: se mueve a la izquierda y para cuando encuentra un '#'
- $D_{\overline{\#}}$: se mueve a la derecha y para cuando encuentra un símbolo distinto a un '#'
- $I_{\overline{\#}}$: se mueve a la izquierda y para cuando encuentra un símbolo distinto a un '#'

MT - Combinando... **ejemplo**

Ejemplo:

Máquina que copia. C: #w# $\vdash_M^*$ #w#w#



MT - Combinando... **mejoras**

- ¿Podemos llegar a **reconocer** el LN con MT's?

MT - Combinando... **mejoras**

- ¿Podemos llegar a **reconocer** el LN con MT's?**... extendamos su concepto** ocupando:

    - cinta infinita para ambos lados
    - varias cintas en vez de una sola
    - varios cabezales en vez de uno
    - una cinta bidimensional
    - **.... y sus combinaciones....**

MT - Combinando... **mejoras**

- Sin embargo, llegamos a....

MT - Combinando... **mejoras**

- Sin embargo, llegamos a…. **nada**.

MT - Combinando... **mejoras**

- Sin embargo, llegamos a…. **nada**.
- Las extensiones de MT’s reconocen los mismos lenguajes que las MT’s ya estudiadas.
- Cualquiera de las extensiones descritas puede ser simulada por una MT “convencional”.
  - En otras palabras, existe una MT convencional por cada MT con extensiones (incluso combinadas).
  - Una MT con extensiones aporta en una mejora del tiempo de ejecución, pero no aporta en reconocer un lenguaje más complejo.

MT - Combinando... **mejoras**

- Hasta ahora, el determinismo ha sido nuestro aliado

- El no-determinismo no aumenta el poder del autómata

MT - Combinando... **mejoras**

- Hasta ahora, el determinismo ha sido nuestro aliado

- El no-determinismo no aumenta el poder del autómata (reconoce el mismo lenguaje)

MT - Combinando... **mejoras**

- Hasta ahora, el determinismo ha sido nuestro aliado
- El no-determinismo no aumenta el poder del autómata (reconoce el mismo lenguaje)

pero …

- ¿Qué pasa si agregamos el ingrediente del no-determinismo a las Máquinas de Turing?

MT - Combinando... **mejoras**

- Hasta ahora, el determinismo ha sido nuestro aliado
- El no-determinismo no aumenta el poder del autómata (reconoce el mismo lenguaje)

pero ...

- ¿Qué pasa si agregamos el ingrediente del no-determinismo a las Máquinas de Turing?

Obtenemos MT no-deterministas.

MT - **No deterministas**

Definición: Una MTND es una cuádrupla (Q, Σ, Δ, s), donde Q, Σ y s son definidos como en las MTD, y Δ es un subconjunto de:

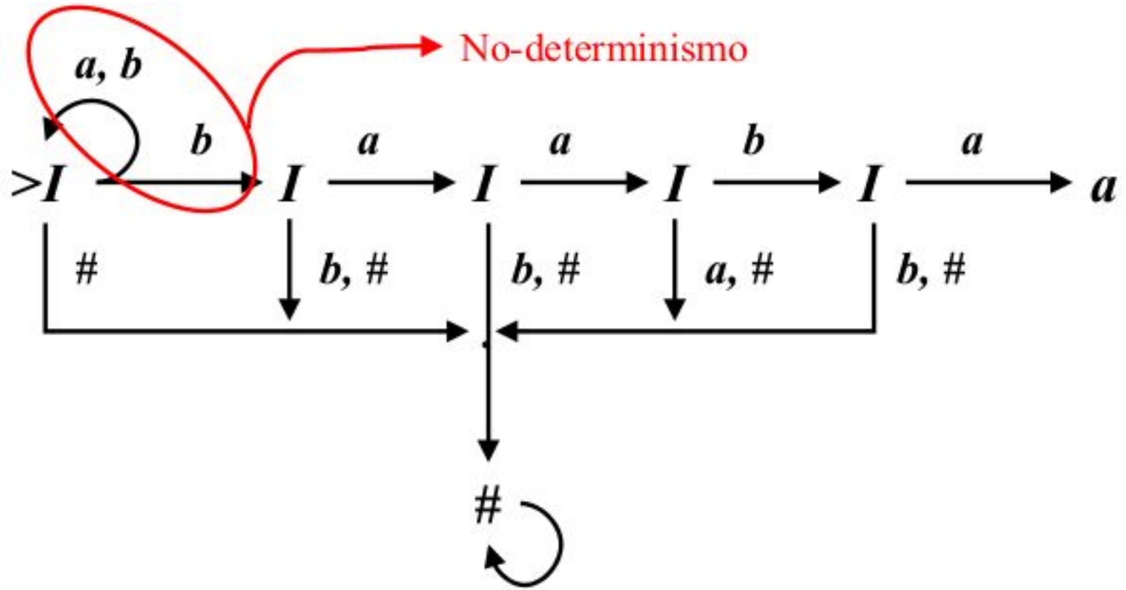
$$(Q \times \Sigma) \times ((Q \cup \{h\}) \times (\Sigma \cup \{I, D\}))$$

- Ahora $\vdash_M$ puede tener varios valores. Podemos ir de una configuración a varias otras en un mismo paso.
- Debemos tener cuidado con la computación de las MTND.
- Consideraremos a las MTND sólo como **aceptadores**
    - Las computaciones de las MTND son erráticas, produciendo diferentes resultados para distintas instancias de ejecución.

MT - **No deterministas**

Ejemplo:

Sea L = {w en {a, b}* / w contiene una ocurrencia del substring abaab}



MT - **No deterministas**

- ¿ganamos algo?

MT - **No deterministas**

- ¿ganamos algo?
    - **no**

MT - **No deterministas**

- ¿ganamos algo?
  - **no**
  - Los lenguajes aceptados por MTND no difieren de los aceptados por MTD.
  - Para cada MTND M_1, se puede construir una MTD M_2 tal que para cualquier palabra w que no contiene blancos:
    - Si M_1 para con entrada w, entonces M_2 para con entrada w
    - Si M_1 no para con entrada w, entonces M_2 no para con entrada w.

Teorema: cualquier lenguaje aceptado por una MTND es aceptado por una MTD.

MT - **En general...**

Las MT se parecen mucho a los computadores:

Máquina de Turing	**Computador**
Maneja una cantidad finita de símbolos	Maneja cantidades finitas de bits o bytes
Escribe/lee símbolos de una cinta	Escribe/lee bits o bytes de Memoria principal o secundaria
Posee un cabezal móvil	En la CPU hay un registro para mantener la instrucción en ejecución: Program Counter
Posee un conjunto de estados que describen su operación	Ejecutan programas (secuencias de instrucciones)

MT - **En general...**

- Una MT es una de las primeras bases para realizar computación.
  - Esta base ha sido implementada y se conoce como computador.
- Somos informáticos, diseñamos/implementamos algoritmos:
  - ¿qué es un algoritmo?
  - ¿qué puede representar un algoritmo?
  - ¿existe una definición matemática de lo que es un algoritmo?

MT - **Tesis de Church**

- Alonzo Church ofrece una definición que conoceremos como Tesis de Church-Turing o simplemente Tesis de Church (TCh).

Todo algoritmo puede representarse como una Máquina de Turing, en caso contrario, no es algoritmo

MT - **Tesis de Church**

- Alonzo Church ofrece una definición que conoceremos como Tesis de Church-Turing o simplemente Tesis de Church (TCh).

Todo algoritmo puede representarse como una Máquina de Turing, en caso contrario, no es algoritmo

- Las máquinas de Turing realmente capturan la noción de lo que es un algoritmo o un procedimiento efectivo llevado a cabo por un humano o por una máquina.

MT - **Tesis de Church**

- Alonzo Church ofrece una definición que conoceremos como Tesis de Church-Turing o simplemente Tesis de Church (TCh).

Todo algoritmo puede representarse como una Máquina de Turing, en caso contrario, no es algoritmo

- Las máquinas de Turing realmente capturan la noción de lo que es un algoritmo o un procedimiento efectivo llevado a cabo por un humano o por una máquina.

- Surgen interesantes preguntas:
  - ¿qué es todo aquello que puede representar un código informático?
  - ¿la actividad cerebral puede ser simulada por una MT?

- La Tesis de Church se cree que es verdadera.

MT - **En resumen**

- Distintos tipos de MT's
- Tristemente son todas equivalentes
- En particular, las MT's que vimos son diseñadas de acuerdo al problema que solucionan
  - ¿Existe una MT para cada problema del universo?
- Queremos más flexibilidad…
  - ¿y si le pasamos a una MT como argumento otra MT?
    - ¿sería como tener hardware y software?
    - ¿cómo realizamos este traspaso?
    - ¿cómo una MT recibe a otra como entrada?
    - ¿se puede recibir a sí misma?
    - ¿para qué nos sirve?

MT - **MT Universales**

- Pensemos un poco: ¿Cómo pasamos una MT como argumento a otras MT's?

MT - **MT Universales**

- Pensemos un poco: ¿Cómo pasamos una MT como argumento a otras MT's?
  - MT M= (Q, Σ, δ, s) con Q y Σ finitos
  - Podemos escribir M usando Q y Σ además de paréntesis, comas, llaves, etc.

MT - **MT Universales**

- Pensemos un poco: ¿Cómo pasamos una MT como argumento a otras MT's?
  - MT M= (Q, Σ, δ, s) con Q y Σ finitos
  - Podemos escribir M usando Q y Σ además de paréntesis, comas, llaves, etc.
  - Sin embargo, esto produce "*palabras*" que no están en el "*lenguaje*" para describir a M
    - M=()}{q_0,,{{q b , c c},)(() no es una MT

MT - **MT Universales**

- Pensemos un poco: ¿Cómo pasamos una MT como argumento a otras MT's?
  - MT M= (Q, Σ, δ, s) con Q y Σ finitos
  - Podemos escribir M usando Q y Σ además de paréntesis, comas, llaves, etc.
  - Sin embargo, esto produce "*palabras*" que no están en el "*lenguaje*" para describir a M
    - M=()}{q_0,,{{q b , c c},)(() no es una MT

      Solución: codificar los estados y símbolos de M

MT - **MT Universales**

- Pensemos un poco: ¿Cómo pasamos una MT como argumento a otras MT's?
  - MT M= (Q, Σ, δ, s) con Q y Σ finitos
  - Podemos escribir M usando Q y Σ además de paréntesis, comas, llaves, etc.
  - Sin embargo, esto produce "*palabras*" que no están en el "*lenguaje*" para describir a M
    - M=(}{q_0,,{{q b , c c},)(() no es una MT

      Solución: codificar los estados y símbolos de M

    - Consideraremos los siguientes conjuntos infinitos y contables:

      $Q_\infty = \{q_1, q_2, q_3, ...\}$, $\Sigma_\infty = \{a_1, a_2, a_3, ...\}$ tal que para cada MT, Q y Σ son subconjuntos de Q_∞ y Σ_∞, respectivamente.

MT - **MT Universales**

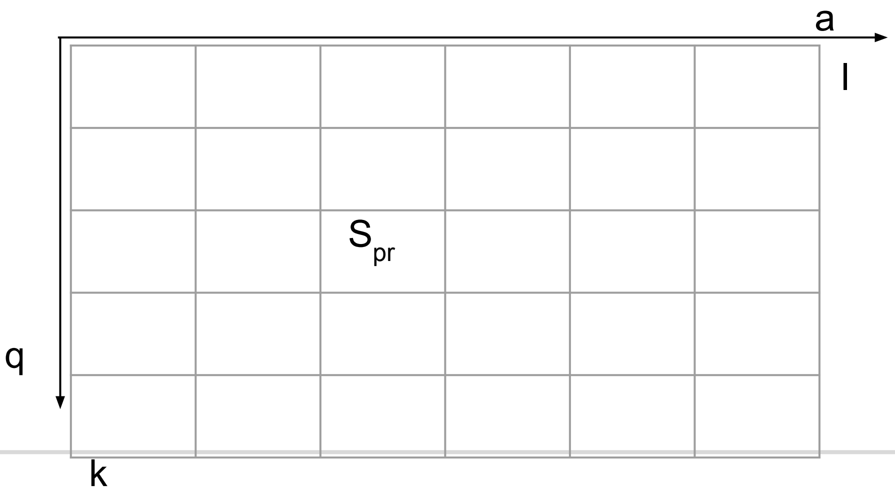
- Adoptaremos la **correspondencia ρ** (codificación) entre los componentes de la MT y las palabras sobre el alfabeto $\{i\}$.

σ	$\rho(\sigma)$
q_k	i^{k+1}
h	i
I	i
D	ii
a_k	i^{k+2}

Sea c otro símbolo, luego codificaremos las MT's sobre el alfabeto {i, c}

MT - **MT Universales**

- Q podemos escribirlo como
  - $Q = \{q_{i1}, q_{i2}, q_{i3}, ..., q_{ik}\}$; con $i_1 < i_2 < i_3 < ... < i_k$ y $Q \subseteq Q_\infty$
- Σ podemos escribirlo como
  - $\Sigma = \{a_{j1}, a_{j2}, a_{j3}, ..., a_{jl}\}$; con $j_1 < j_2 < j_3 < ... < j_l$ y $\Sigma \subseteq \Sigma_\infty$
- Definimos *k* · *l* palabras S_{pr} ($1 \leq p \leq k$ y $1 \leq r \leq l$)
- Cada S_{pr} codifica el valor de la función de transición sobre un par estado-símbolo, llamado par (q_{ip}, a_{jr}).



MT - **MT Universales**

- Específicamente:
  - Sea la transición $\delta\,(q_{ip}, a_{jr}) = (q', b)$ con q' en $Q \cup \{h\}$ y b en $\Sigma \cup \{I, D\}$, entonces

$$S_{pr} = cw_1cw_2cw_3cw_4c$$

donde

$$w_1 = \rho(q_{ip})$$
$$w_2 = \rho(a_{jr})$$
$$w_3 = \rho(q')$$
$$w_4 = \rho(b)$$

codificación de la máquina M

  - Finalmente, $\rho(M)$ se escribe

$$cS_0cS_{11}S_{12}\ldots S_{1l}S_{21}S_{22}\ldots S_{2l}\ldots S_{k1}S_{k2}\ldots S_{kl}c$$

donde $S_0 = \rho(s)$

codificación del estado inicial

MT - **MT Universales**

- Tenemos definida la MTU
- Podemos imaginarla como un computador al que le cargamos programas (ρ(M)) y datos de entrada al programa (ρ(w))
- Es claro que:

Cualquier programa, en cualquier lenguaje de programación, traducido por cualquier compilador y ejecutado en cualquier computador = Máquina de Turing

- Comencemos a resolver problemas con MTU
  - Reconozcamos lenguajes regulares
  - Reconozcamos lenguajes independientes al contexto
  - Calculemos series de Fourier...

MT - **MT Universales**

¿Podemos calcular o reconocer cualquier cosa?

MT - **MT Universales**

¿Podemos calcular o reconocer cualquier cosa?

Inspiración:

¿Qué hay más, números naturales o reales?

MT - **MT Universales**

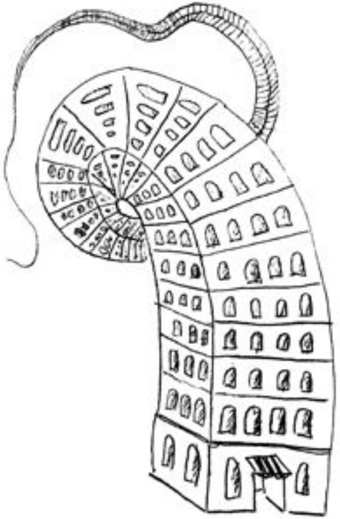
¿Podemos calcular o reconocer cualquier cosa?

Inspiración:

¿Qué hay más, números naturales o reales?

Entre 0 y 2 ¿cuántos reales existen? ¿cuántos naturales existen?

MT - **MT Universales**



¿Podemos calcular o reconocer cualquier cosa? **NO**

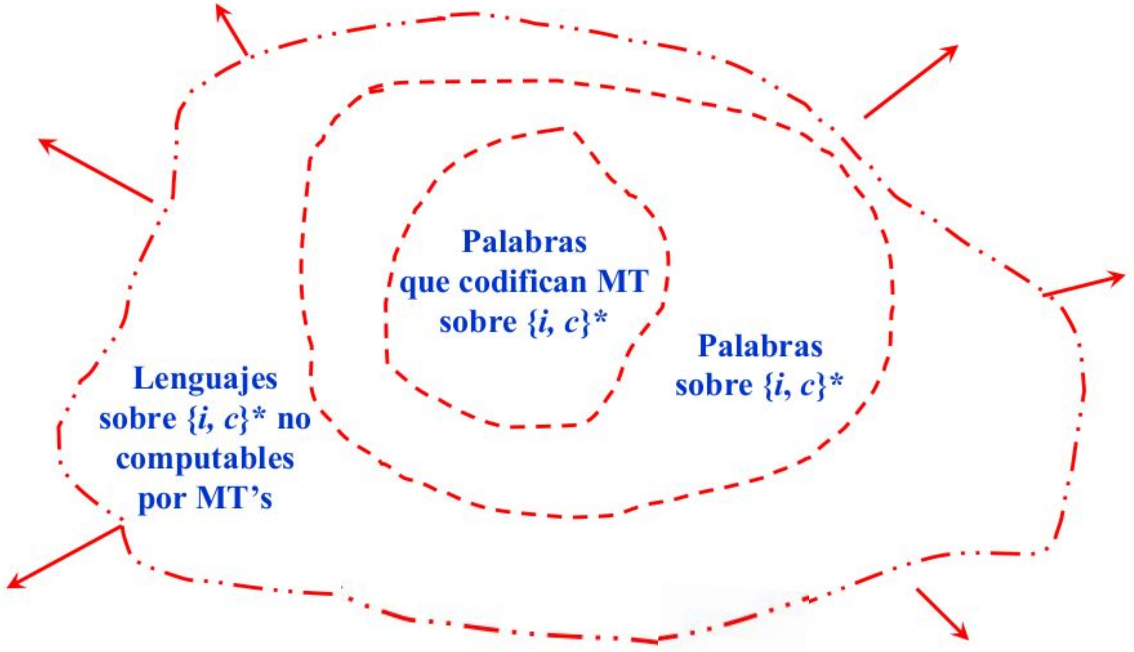
Inspiración:

¿Qué hay más, números naturales o reales?

Entre 0 y 2 ¿cuántos reales existen? ¿cuántos naturales existen?

- Existen **infinitos-contables** palabras sobre {c, i}* y no todas ellas codifican a MT's.
- Por lo tanto, existen infinitos-contables MT's
- Por otro lado, hay **infinitos-incontables** lenguajes sobre {c, i}*
- Debe haber lenguajes que MT no pueden computar

MT - MT Universales, **informalmente**



Hemos visto que...

- Simplemente hay problemas que las MT's no pueden resolver, ya que **no** existe una MT asociada a ese lenguaje.

Hemos visto que...

- Simplemente hay problemas que las MT's no pueden resolver, ya que **no** existe una MT asociada a ese lenguaje.

- En otras palabras ***no existe un algoritmo que los resuelva***.

Problema de correspondencia de Post

- Fue propuesto por **Emil Post**.
- Es un problema de decisión.
- Informalmente, el problema consiste en:

Dado un diccionario bilingüe que contiene pares de frases, es decir, listas de palabras, que significan lo mismo, se debe decidir si existe una frase que significa lo mismo en ambos lenguajes.

Problema de correspondencia de Post

Definición del problema:

Dadas dos listas finitas de palabras $u_1, \ldots, u_n$ y $v_1, \ldots, v_n$ sobre el alfabeto Σ, una solución al problema es una secuencia de índices
$i_1, \ldots, i_k$; $1 \leq i_j \leq n$, tales que $u_{i1}, \ldots, u_{ik} = v_{i1}, \ldots, v_{ik}$

Así, el problema consiste en saber si existe una solución para el problema planteado.

Ejemplo

u_1	u_2	u_3	u_4
aba	bbb	aab	bb

$\stackrel{?}{=}$

v_1	v_2	v_3	v_4
a	aaa	$abab$	$babba$

Problema de correspondencia de Post

Ejemplo,

u_1	u_2	u_3	u_4
aba	bbb	aab	bb

v_1	v_2	v_3	v_4
a	aaa	$abab$	$babba$

Problema de correspondencia de Post

Ejemplo,

u_1	u_2	u_3	u_4
aba	bbb	aab	bb

v_1	v_2	v_3	v_4
a	aaa	$abab$	$babba$

Una posible solución es la secuencia 1, 4, 3, 1

$$\begin{aligned} u_1 u_4 u_3 u_1 &= aba + bb + aab + aba \\ &= ababbaababa \\ &= a + babba + abab + a \\ &= v_1 v_4 v_3 v_1 \end{aligned}$$

Problema de correspondencia de Post

Ejemplo,

u_1	u_2	u_3	u_4
aba	bbb	aab	bb

v_1	v_2	v_3	v_4
a	aaa	$abab$	$babba$

Una posible solución es la secuencia 1, 4, 3, 1

$$\begin{aligned} u_1 u_4 u_3 u_1 &= aba + bb + aab + aba \\ &= ababbaababa \\ &= a + babba + abab + a \\ &= v_1 v_4 v_3 v_1 \end{aligned}$$

y si u_4 y v_4 no estuvieran?

Otro ejemplo...

Escribamos un algoritmo que genere como salida **todos** los códigos Java que el compilador rechazará por problemas de sintaxis.

Otro ejemplo...

Escribamos un algoritmo que genere como salida **todos** los códigos Java que el compilador rechazará por problemas de sintaxis.

¿Podemos realizar esto de manera sistemática?

Otro ejemplo...

Escribamos un algoritmo que genere como salida **todos** los códigos Java que el compilador rechazará por problemas de sintaxis.

¿Podemos realizar esto de manera sistemática?

¿Cuál es el problema para escribir tal algoritmo?

debemos estudiar problemas no decidibles…

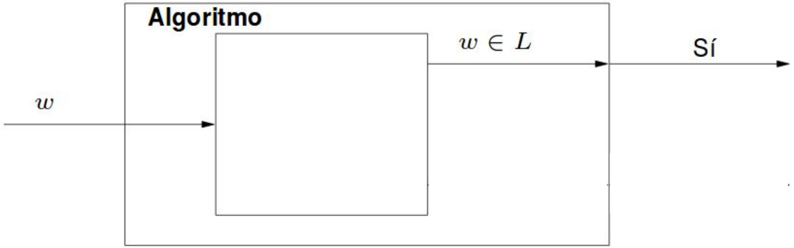
Hasta ahora...

- Hemos visto que si L es un lenguaje decidible, existe una MT que siempre se detiene si una palabra cualquiera w pertenece a L.

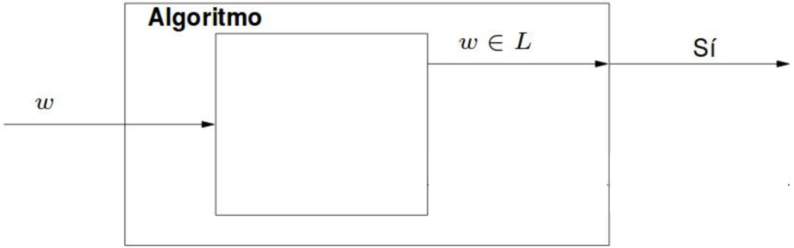
- Los lenguajes decidibles también se conocen como **lenguajes recursivos**.

- Sin embargo, existe una calificación menos restrictiva:
    - **Definición**: Un lenguaje L es recursivamente enumerable si existe una MT M que cuando es alimentada con alguna palabra w perteneciente a L, se detiene diciendo SÍ.
        - El comportamiento **no está definido** cuando w no está en L

Lenguajes recursivamente enumerables

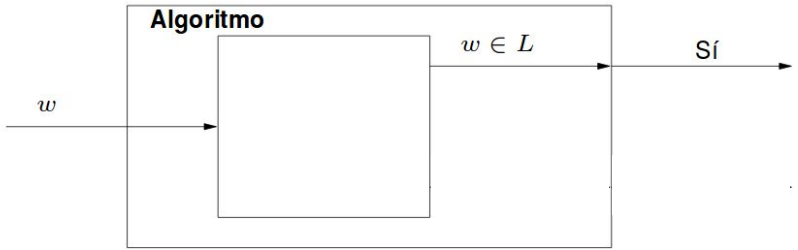


Lenguajes recursivamente enumerables

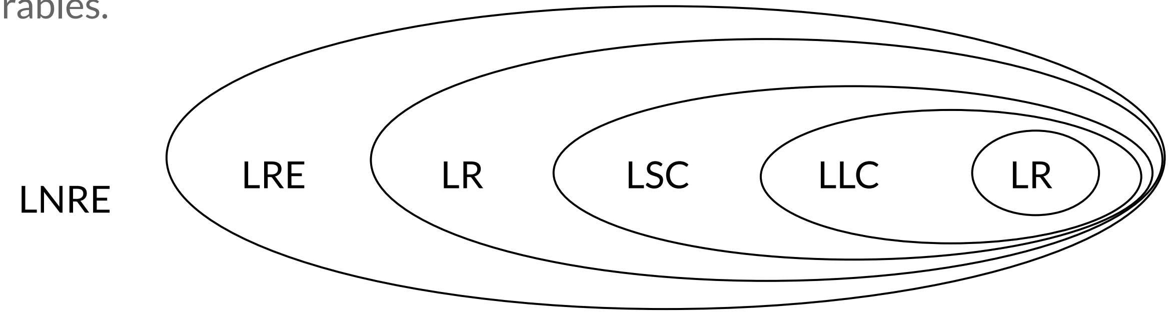


- Todo lenguaje recursivo (decidible) es recursivamente enumerable.

Lenguajes recursivamente enumerables



- Todo lenguaje recursivo (decidible) es recursivamente enumerable.
  - Es fácil ver que un lenguaje recursivo es un caso particular de los lenguajes recursivamente enumerables.



¿Por qué enumerable?

¿Por qué enumerable?

- Enumerable tiene que ver con enlistar “algo”

¿Por qué enumerable?

- Enumerable tiene que ver con enlistar "algo"

L es LRE $\Rightarrow$ MT que corre sin entrada y genera todas las palabras de L

¿Por qué enumerable?

- Enumerable tiene que ver con enlistar "algo"

L es LRE $\rightleftarrows$ MT que corre sin entrada y genera todas las palabras de L

¿Por qué enumerable?

- Enumerable tiene que ver con enlistar “algo”

L es LRE $\rightleftarrows$ MT que corre sin entrada y genera todas las palabras de L

- Si L es infinito, entonces la MT no se detiene.

¿Por qué enumerable?

- Enumerable tiene que ver con enlistar "algo"

L es LRE $\rightleftarrows$ MT que corre sin entrada y genera todas las palabras de L

- Si L es infinito, entonces la MT no se detiene.
- Garantía: cada w de L es generada en tiempo finito

Las gramáticas también juegan

- Si un lenguaje L es decidible, entonces hay una MT que enumera todas las palabras de L
- Podemos ver que este es el mismo comportamiento de las gramáticas, luego se pueden producir todas las palabras del lenguaje.
- Así, diremos que un lenguaje L es **Turing-decidible** (o recursivo) si y sólo si existe una gramática que genere L.

- LRegulares
- LLC
- LSC
- LRecursivos

Son Turing-Decidibles

decidible e indecidible

Halting Problem

- **Primer problema demostrado como indecidible**.

- Formulado por Alan Turing en 1936.

- Informalmente:
  "Dada una descripción de un programa P y una entrada x para éste, se debe determinar si P se detiene (termina) dada la entrada x"

- Formalmente:
  - Sea M una MT arbitraria con alfabeto de entrada Σ_0. Sea w en Σ_0^*. ¿M para con entrada w?

HP = {ρ(M)ρ(w) | M acepta la entrada w}

Si existiera una manera de determinar cuando M no va a detenerse, podríamos rechazar la entrada...

Halting Problem - **indecidible**

- Para demostrar que el HP es indecidible, debemos probar la siguiente afirmación:

  NO existe una Máquina de Turing M_h, que tomando como entrada cualquier máquina M_0, se detenga después de un tiempo finito y …

  - responda SÍ cuando M_0 se detenga, y…
  - responda NO cuando M_0 no se detenga.

Halting Problem - **indecidible**

- Para demostrar que el HP es indecidible, debemos probar la siguiente afirmación:

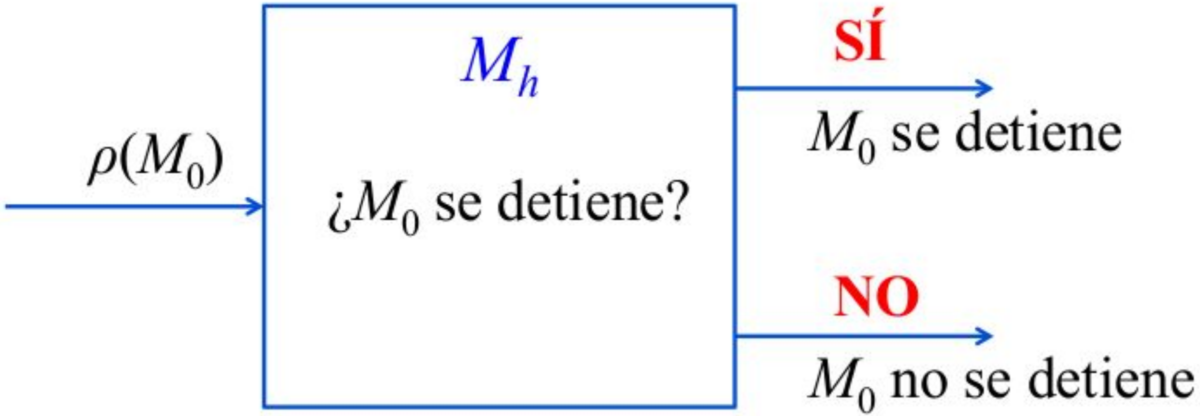
  NO existe una Máquina de Turing M_h, que tomando como entrada cualquier máquina M_0, se detenga después de un tiempo finito y ...

  - responda SÍ cuando M_0 se detenga, y...
  - responda NO cuando M_0 no se detenga.

Si M_h **existiese**, el HP sería decidible...

Halting Problem - **indecidible**

M_h sería así...



Halting Problem - **indecidible**

Estrategia de la demostración

- Demostraremos que no existe M_h que decide HP.

Halting Problem - **indecidible**

Estrategia de la demostración (Buscamos una ***contradicción***)

- Demostraremos que no existe M_h que decide HP.
- Hipótesis: **M_h existe**.

Halting Problem - **indecidible**

Estrategia de la demostración (Buscamos una ***contradicción***)

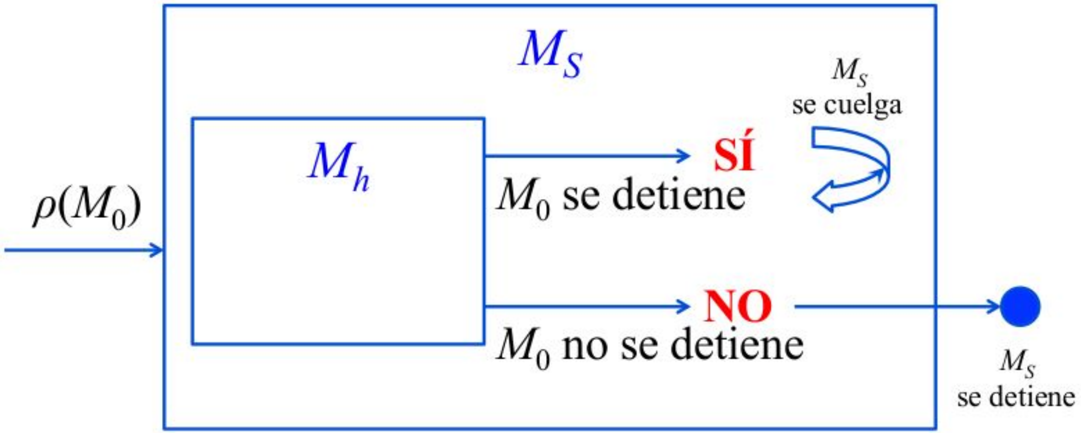
- Demostraremos que no existe M_h que decide HP.
- Hipótesis: **M_h existe**.

Entonces...

- Construyamos una nueva MT M_s que se comporte de la siguiente manera:
  - Tome como entrada M_0.
  - Simule M_h y le entregue como entrada M_0.
    - Por hipótesis, M_h siempre se detiene respondiendo SÍ o NO, según M_0 se detiene o no.
  - Si M_h responde SÍ, entonces M_s no se detiene,
  - Si M_h responde NO, entonces M_s se detiene.

Halting Problem - **indecidible**

- Si M_0 **se detiene**, entonces M_S **no se detiene**.
- Si M_0 **no se detiene**, entonces M_S **se detiene**.



Halting Problem - **indecidible**

Si M_s es entrada de si misma…

Halting Problem - **indecidible**

Si M_S es entrada de si misma…

- Existe una entrada para la cual M_S produce una contradicción.
- Si la entrada de M_S es M_S ($M_0 = M_S$), se tiene el siguiente comportamiento:

$$\boxed{\begin{array}{l} M_S \text{ se detiene} \Rightarrow M_S \text{ no se detiene} \\ \\ M_S \text{ no se detiene} \Rightarrow M_S \text{ se detiene} \end{array}}$$

Halting Problem - **indecidible**

Si M_S es entrada de si misma…

- Existe una entrada para la cual M_S produce una contradicción.
- Si la entrada de M_S es M_S ($M_0 = M_S$), se tiene el siguiente comportamiento:

$$\boxed{\begin{array}{l} M_S \text{ se detiene} \Rightarrow M_S \text{ no se detiene} \\ \\ M_S \text{ no se detiene} \Rightarrow M_S \text{ se detiene} \end{array}}$$

Para entender esta contradicción, veamos dos casos:

Caso 1: M_S se detiene cuando es entrada de si misma.

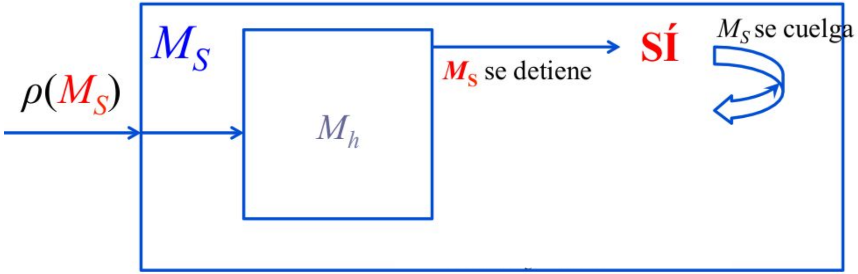
Caso 2: M_S no se detiene cuando es entrada de si misma.

Halting Problem - **indecidible**

Caso 1: M_S **se detiene** cuando es entrada de si misma.

I. M_S es entregada como su propia entrada (una copia)
II. M_h recibe como entrada a M_S.
III. Por hipótesis, M_h responderá SÍ ya que **M_S se detiene**.
IV. Una vez que M_h responde SÍ, M_S se cuelga.

Esto conlleva a que el supuesto de que M_S se detiene al aplicarse a sí misma, implica que M_S no se detiene.

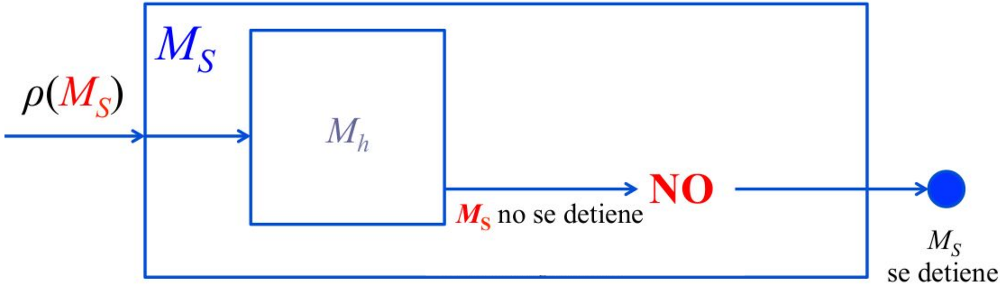


Halting Problem - **indecidible**

Caso 2: **M_S no se detiene** cuando es entrada de si misma.

I. M_S es entregada como su propia entrada (una copia).
II. M_h recibe como entrada a M_S.
III. Por hipótesis, M_h responderá NO ya que **M_S no se detiene**.
IV. Una vez que M_h responde NO, M_S se detiene.

Esto conlleva a la suposición de que M_S no se detiene al aplicarse a sí misma, lo que implica que M_S se detiene.



Halting Problem - **indecidible**

M_S se detiene ⇒ M_S no se detiene

M_S no se detiene ⇒ M_S se detiene

CONTRADICCIÓN

- Para los dos casos, es imposible tanto que M_S termine como que no termine
- Ya que M_S fue construida acorde a las reglas determinadas, la única responsable de la contradicción es la hipotética M_h.

Halting Problem - **indecidible**

M_S se detiene ⇒ M_S no se detiene

M_S no se detiene ⇒ M_S se detiene

CONTRADICCIÓN

- Para los dos casos, es imposible tanto que M_S termine como que no termine
- Ya que M_S fue construida acorde a las reglas determinadas, la única responsable de la contradicción es la hipotética M_h.
- Por lo tanto M_h no existe y el *Halting Problem* es **INDECIDIBLE**.

Si existiera una manera de determinar cuando M no va a detenerse, podríamos rechazar la entrada...

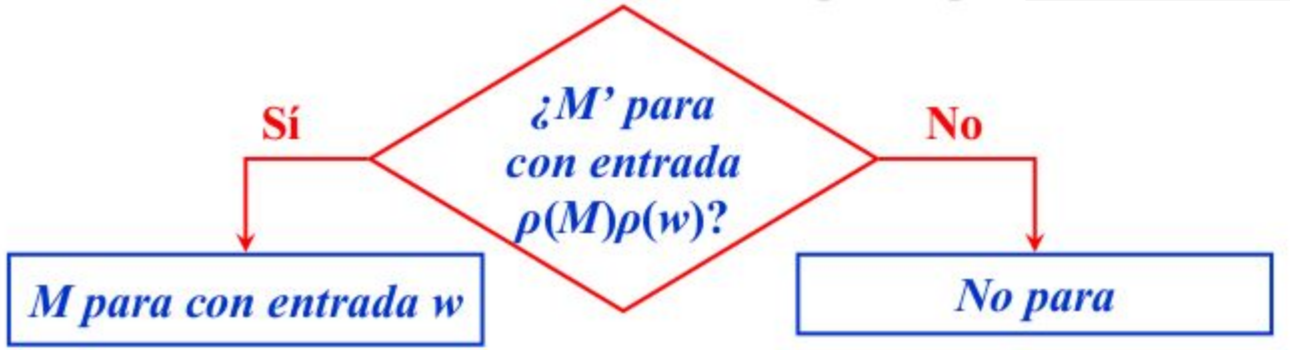
sin embargo, NO ES POSIBLE...

Halting Problem - **Clase**

Teorema: **HP** es recursivamente enumerable.

Demostración:

Solamente necesitamos una MT **M'** que acepte **HP**:



Claramente **M'** es una **MTU**

Definición:

Un lenguaje L es recursivamente enumerable si existe una MT M que cuando es alimentada con alguna palabra w perteneciente a L, se detiene diciendo SÍ.

Halting Problem - **Clase**

El Halting Problem no sólo es RE... HP es RE-Completo

Definimos:

- Sea L un lenguaje RE cualquiera sobre un alfabeto Σ cualquiera.
- Sea M_L la MT que acepta L
- Sea M_{HP} una MT que decide el Halting Problem
- Sea $w \in \Sigma^*$

M_{HP} es alimentada con $\rho(M_L)\rho(w)$

- Si M_{HP} responde √, entonces M_L acepta (se detiene) y $w \in L$
- Si M_{HP} responde ×, entonces M_L no acepta (no se detiene) y $w \notin L$

En otras palabras, M_{HP} decide L

Halting Problem - **Clase**

Ya que L es un LRE cualquiera, **si el Halting Problem se decide**, entonces todos los LRE serían decidibles (i.e. Recursivos).

noción de Completitud

Complejidad Computacional

UdeSantiago de Chile

LABOR LÆTITIA NOSTRA

Complejidad Computacional

- Hay problemas o lenguajes que no pueden ser decididos por una MT (no existe un algoritmo que los resuelva).

Complejidad Computacional

- Hay problemas o lenguajes que no pueden ser decididos por una MT (no existe un algoritmo que los resuelva).

- Por otro lado, tenemos problemas que sí tienen solución.

Complejidad Computacional

- Hay problemas o lenguajes que no pueden ser decididos por una MT (no existe un algoritmo que los resuelva).

- Por otro lado, tenemos problemas que sí tienen solución.

¿Qué un problema tenga solución significa que el problema es fácil?

Complejidad Computacional

- Hay problemas o lenguajes que no pueden ser decididos por una MT (no existe un algoritmo que los resuelva).

- Por otro lado, tenemos problemas que sí tienen solución.

¿Qué un problema tenga solución significa que el problema es fácil?

¿Qué significa que un problema sea fácil?

Complejidad Computacional

- Hay problemas o lenguajes que no pueden ser decididos por una MT (no existe un algoritmo que los resuelva).

- Por otro lado, tenemos problemas que sí tienen solución.

¿Qué un problema tenga solución significa que el problema es fácil?

¿Qué significa que un problema sea fácil?

¿Qué significa que un problema sea difícil?

Complejidad Computacional

Scheduling

"el arte de asignar recursos a tareas con el fin de asegurar el término de ellas en un plazo razonable de tiempo"

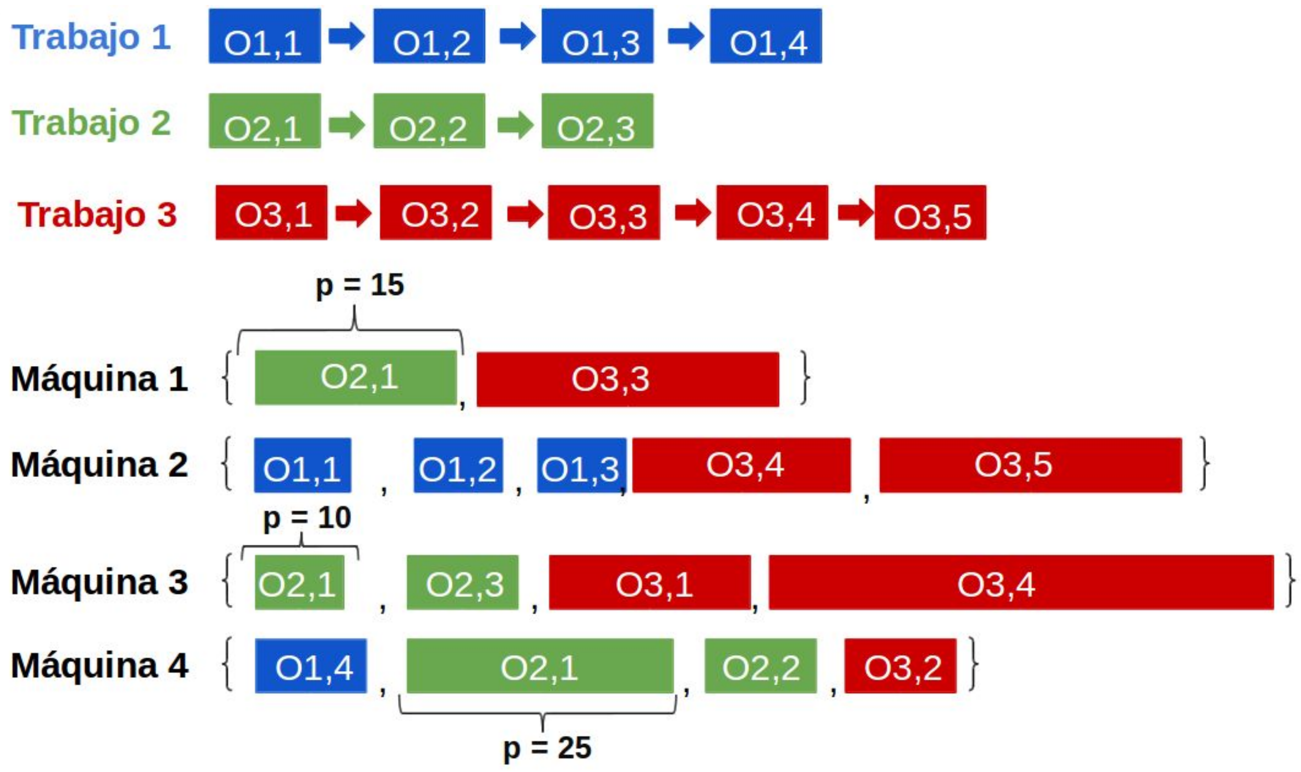
Complejidad Computacional

Scheduling

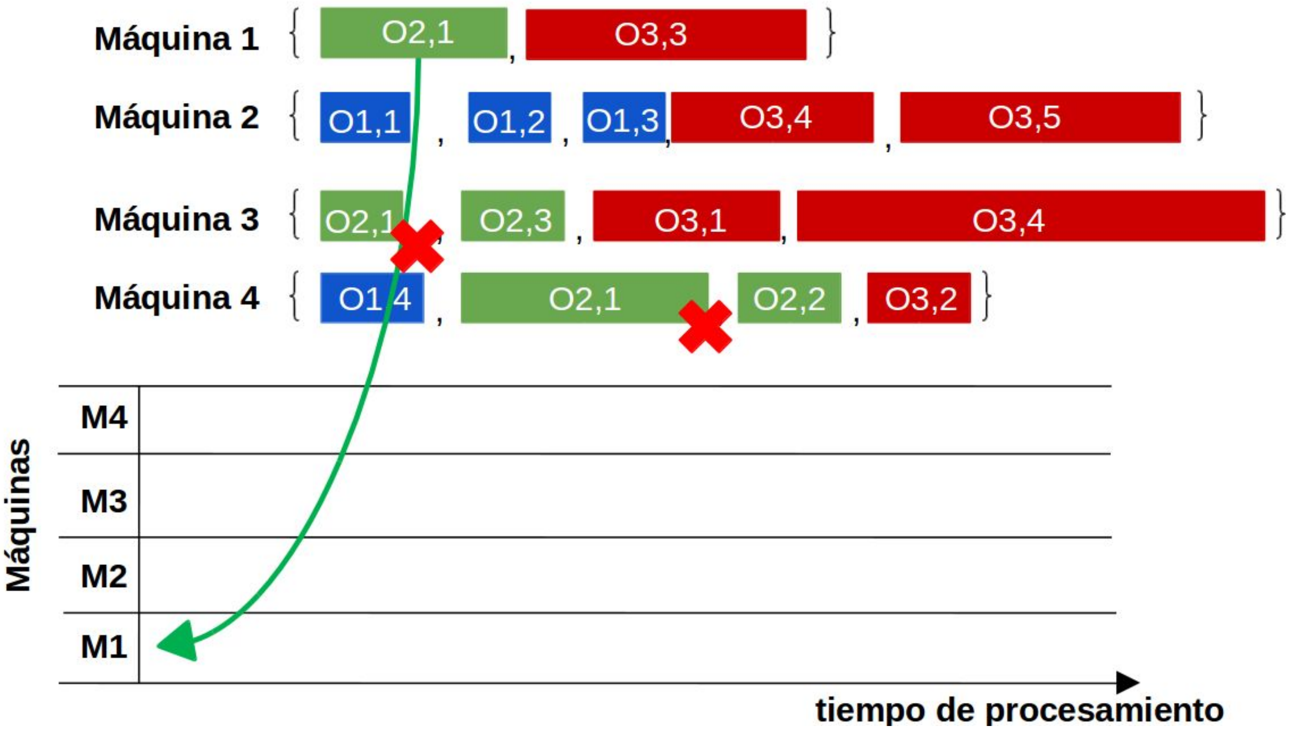
Se compone de un grupo de trabajos, divididos en operaciones y un grupo de máquinas, sobre las que las operaciones de los trabajos deben ser procesadas.

En particular, el **Flexible Job shop Scheduling Problem** (FJSP), corresponde a un problema de scheduling, donde una máquina es capaz de procesar más de un tipo de operación

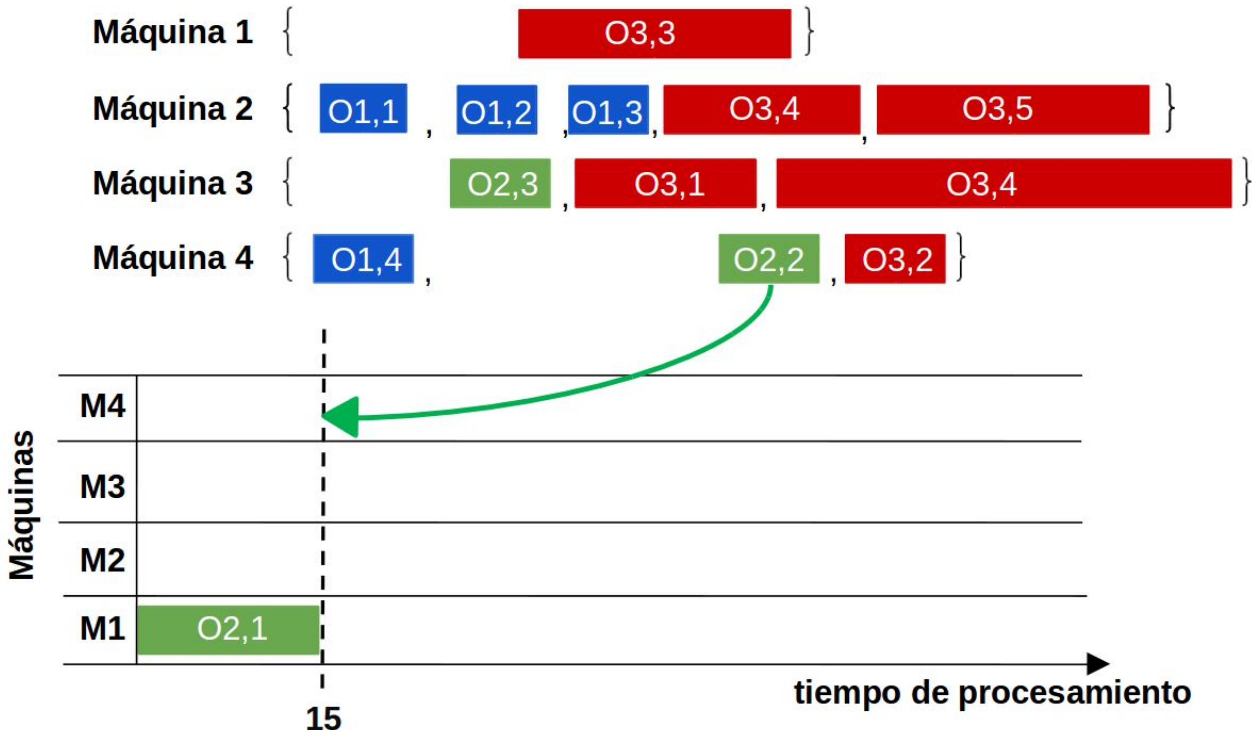
Scheduling - **FJSP**



Scheduling - **FJSP**



Scheduling - **FJSP**

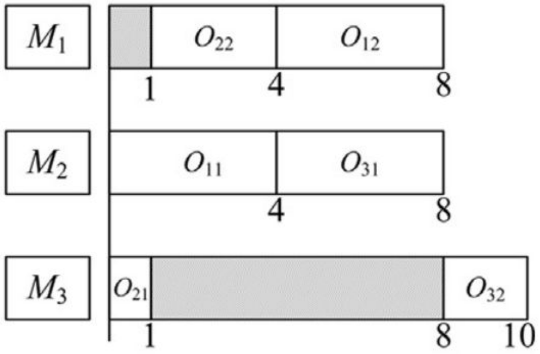


Complejidad Computacional

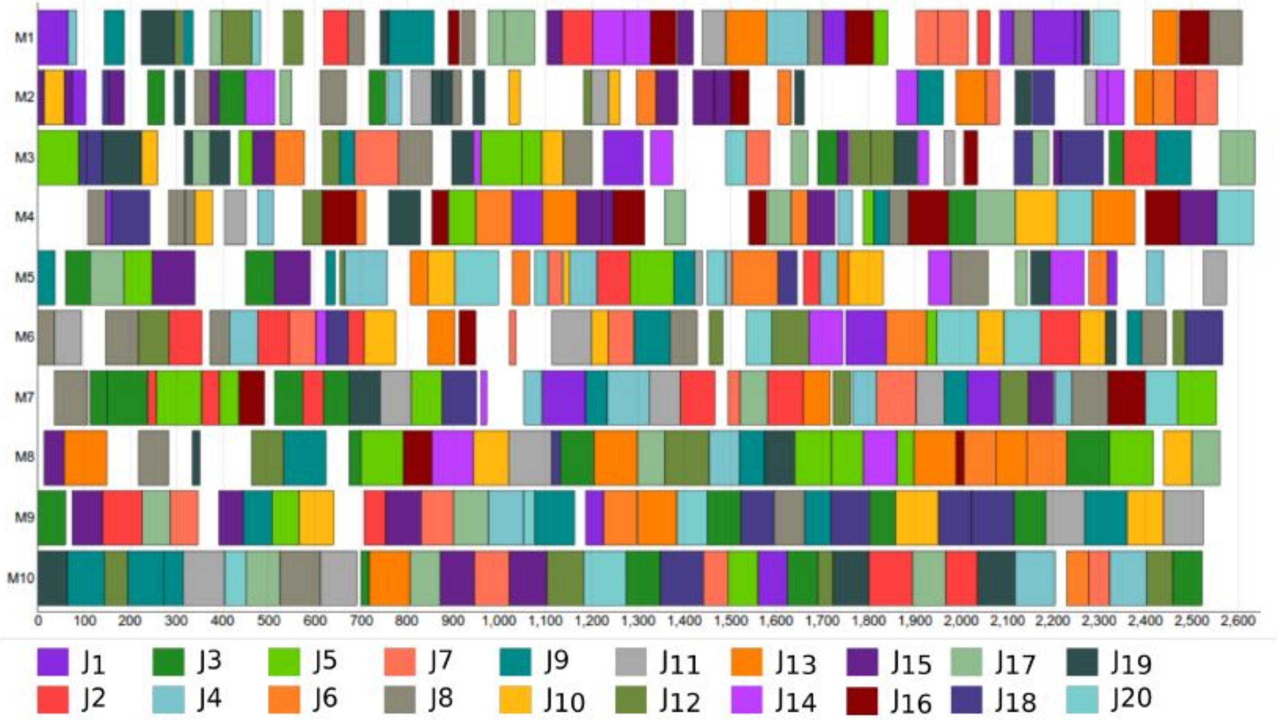
Scheduling - Flexible Job shop Scheduling Problem (FJSP)

Problema “pequeño”

Job	Ope	Alternative machines with processing time		
		M_1	M_2	M_3
J_1	O_{11}	5	4	4
	O_{12}	4	2	3
J_2	O_{21}	2	2	1
	O_{22}	3	4	3
J_3	O_{31}	5	4	5
	O_{32}	2	-	2



Scheduling - **FJSP**



Carta Gantt instancia 18a Dauzère-Pérèz, algoritmo EE-A10

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.
- Cuando empleamos el término complejo lo asociamos habitualmente con eficiencia.

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.
- Cuando empleamos el término complejo lo asociamos habitualmente con eficiencia.

Definimos un algoritmo A y un problema P, entonces,

$$\nexists A \text{ eficiente, entonces } P \text{ es complejo}$$

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.
- Cuando empleamos el término complejo lo asociamos habitualmente con eficiencia.

Definimos un algoritmo *A* y un problema *P*, entonces,

$\nexists$ *A* eficiente, entonces *P* es complejo

¿ $\exists$ *A* no es eficiente, entonces *P* es complejo?

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.
- Cuando empleamos el término complejo lo asociamos habitualmente con eficiencia.

Definimos un algoritmo *A* y un problema *P*, entonces,

$\nexists$ *A* eficiente, entonces *P* es complejo

¿ $\exists$ *A* no es eficiente, entonces *P* es complejo?

Si P es “sencillo” no significa que **todos** los algoritmos que lo resuelven sean eficientes.

Complejidad Computacional

- Hablar de “complejidad” necesita ser formalizado.
- Cuando empleamos el término complejo lo asociamos habitualmente con eficiencia.

Definimos un algoritmo *A* y un problema *P*, entonces,

$\nexists$ *A* eficiente, entonces *P* es complejo

¿ $\exists$ *A* no es eficiente, entonces *P* es complejo?

Si P es “sencillo” no significa que **todos** los algoritmos que lo resuelven sean eficientes.

La tarea fundamental de la teoría de la complejidad computacional es jerarquizar los problemas de acuerdo a su complejidad computacional.

Complejidad Computacional

Enfrentando un nuevo problema...

Complejidad Computacional

Enfrentando un nuevo problema...

¿puede ser resuelto eficientemente?

Complejidad Computacional

Enfrentando un nuevo problema...

¿puede ser resuelto eficientemente?

SI

encontrar un algoritmo eficiente
que lo resuelva

Complejidad Computacional

Enfrentando un nuevo problema...

¿puede ser resuelto eficientemente?

SI

encontrar un algoritmo eficiente
que lo resuelva

NO

según el contexto, ¿vale la pena
intentar resolverlo ineficientemente?

Complejidad Computacional

Enfrentando un nuevo problema...

¿puede ser resuelto eficientemente?

SI

encontrar un algoritmo eficiente que lo resuelva

NO

según el contexto, ¿vale la pena intentar resolverlo ineficientemente?

SI

encontrar un algoritmo ineficiente que lo resuelva

Complejidad Computacional

Enfrentando un nuevo problema...

¿puede ser resuelto eficientemente?

SI

NO

encontrar un algoritmo eficiente que lo resuelva

según el contexto, ¿vale la pena intentar resolverlo ineficientemente?

SI

NO

encontrar un algoritmo ineficiente que lo resuelva

buscar soluciones aproximadas (Optimización)

Complejidad Computacional

- Estableceremos una clasificación **cuantitativa** en términos de esfuerzo de una MT para computar.

Más **esfuerzo** computacional ⇔ Más **pasos** de la MT

Complejidad Computacional

- Estableceremos una clasificación **cuantitativa** en términos de esfuerzo de una MT para computar.

Más **esfuerzo** computacional ⇔ Más **pasos** de la MT

- El problema fundamental de nuestra disciplina:

Dado un problema, encontrar un algoritmo eficiente para solucionarlo o demostrar que no existe tal algoritmo.

Complejidad Computacional

- ¿Por qué creemos que las MT's son una buena formalización del concepto de algoritmo?

Complejidad Computacional

- ¿Por qué creemos que las MT's son una buena formalización del concepto de algoritmo?
  - Porque cada programa de una MT puede ser implementado.

Complejidad Computacional

- ¿Por qué creemos que las MT's son una buena formalización del concepto de algoritmo?
  - Porque cada programa de una MT puede ser implementado.
  - Porque todos los algoritmos conocidos han podido ser implementados en MT.

Complejidad Computacional

- ¿Por qué creemos que las MT's son una buena formalización del concepto de algoritmo?
  - Porque cada programa de una MT puede ser implementado.
  - Porque todos los algoritmos conocidos han podido ser implementados en MT.
  - Porque todos los otros intentos por formalizar este concepto fueron reducidos a las MT.
    - Los mejores intentos resultaron ser equivalentes a MT
    - Todos los intentos "razonables" fueron reducidos eficientemente.

Complejidad Computacional

- ¿Por qué creemos que las MT's son una buena formalización del concepto de algoritmo?
  - Porque cada programa de una MT puede ser implementado.
  - Porque todos los algoritmos conocidos han podido ser implementados en MT.
  - Porque todos los otros intentos por formalizar este concepto fueron reducidos a las MT.
    - Los mejores intentos resultaron ser equivalentes a MT
    - Todos los intentos "razonables" fueron reducidos eficientemente.
- Recordemos que debido a la Tesis de Church:

Algoritmo = Máquinas de Turing

Complejidad Computacional

Esfuerzo computacional…

Dada una entrada w para una MT, definimos una función $T : N \rightarrow N$ tal que $T(|w|)$ es el tiempo o número de pasos (configuraciones) que dicha MT tarda en computar w, en función de su tamaño $|w|$.

Un lenguaje L es decidible en tiempo T si existe una MT determinista (MTD) que decide L en tiempo T.

La clase de todos los lenguajes decidibles en tiempo T se denota **DTIME(T)**.

Complejidad Computacional

- Dado lo anterior, acotamos el número de pasos de una MT por una función del largo de la entrada.

Complejidad Computacional

- Dado lo anterior, acotamos el número de pasos de una MT por una función del largo de la entrada.
- Dadas dos MTDs M_1 y M_2 que deciden un lenguaje L. Sea $w \in L$ con $|w| = n$, supongamos:

$$T_1(n) = n^3 \qquad \text{y} \qquad T_2(n) = 6T_1(n)^2 + 3n + 15$$

Complejidad Computacional

- Dado lo anterior, acotamos el número de pasos de una MT por una función del largo de la entrada.
- Dadas dos MTDs M_1 y M_2 que deciden un lenguaje L. Sea $w \in L$ con $|w| = n$, supongamos:

$$T_1(n) = n^3 \qquad \text{y} \qquad T_2(n) = 6T_1(n)^2 + 3n + 15$$

- Recordemos que las variaciones de MTDs no aumentan el poder computacional.

Complejidad Computacional

- Dado lo anterior, acotamos el número de pasos de una MT por una función del largo de la entrada.
- Dadas dos MTDs M_1 y M_2 que deciden un lenguaje L. Sea $w \in L$ con $|w| = n$, supongamos:

$$T_1(n) = n^3 \qquad y \qquad T_2(n) = 6T_1(n)^2 + 3n + 15$$

- Recordemos que las variaciones de MTDs no aumentan el poder computacional.
- Ambas MTDs poseen un tiempo de ejecución polinomial.

Complejidad Computacional

- Dado lo anterior, acotamos el número de pasos de una MT por una función del largo de la entrada.
- Dadas dos MTDs M_1 y M_2 que deciden un lenguaje L. Sea $w \in L$ con $|w| = n$, supongamos:

$$T_1(n) = n^3 \qquad \text{y} \qquad T_2(n) = 6T_1(n)^2 + 3n + 15$$

- Recordemos que las variaciones de MTDs no aumentan el poder computacional.
- Ambas MTDs poseen un tiempo de ejecución polinomial.
- **No nos interesan** las diferencias “sutiles” entre T_1 y T_2, lo relevante es la tasa de crecimiento.
- Por lo tanto, DTIME(T_1) y DTIME(T_2) deberíamos considerarlas dentro de una misma clase más general.

Complejidad Computacional

Funciones superiormente acotadas: O

- Todo polinomio está superiormente acotado por otro polinomio.
- La tasa de crecimiento de cualquier polinomio puede representarse simplemente por su grado.
- Ej: Si el tiempo de complejidad de un algoritmo es

  $$T(n) = 3n^2 + 6n,$$

  decimos que tiene orden n^2 y usamos la notación $O(n^2)$, tal que $T(n) \in O(n^2)$ o bien, $T(n) = O(n^2)$.

- Formalmente, $f(x) = O(g(x))$ si existen k, x_0 reales positivos tal que $|f(x)| \leq k|g(x)|$, para todo $x \geq x_0$

Complejidad Computacional

Algunas propiedades

- Adición
  - Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$
  - Si f y g son funciones positivas, $f + O(g) = O(f + g)$
- Multiplicación
  - Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 \cdot f_2 = O(g_1 \cdot g_2)$
  - $f \cdot O(g) = O(f \cdot g)$
  - $O(k \cdot g) = O(g)$, para toda constante $k > 0$.

Complejidad Computacional

Otras funciones acotadas: Ω y Θ

- $f(x) = \Omega(g(x))$ si existen k, x_0 reales positivos tales que
  $k|g(x)| \leq |f(x)|$, para todo $x \geq x_0$

- $f(x) = \Theta(g(x))$ si existen k_1, k_2, x_0 reales positivos tales que
  $k_1|g(x)| \leq |f(x)| \leq k_2|g(x)|$, para todo $x \geq x_0$

- Ej: Si $T(n) = 3n^2 + 6n$, entonces:
  - $T(n) \in \Omega(n^2)$ y $T(n) \in \Omega(n)$,
  - $T(n) \in \Theta(n^2)$, pero $T(n) \notin \Theta(n)$ y $T(n) \notin \Theta(n\ 3)$.

- Además existen o y ω, como equivalentes de O y Ω, pero para cotas estrictas (<).

Funciones acotadas

- Funciones superiormente acotadas: O

- Otras funciones acotadas: Ω y Θ

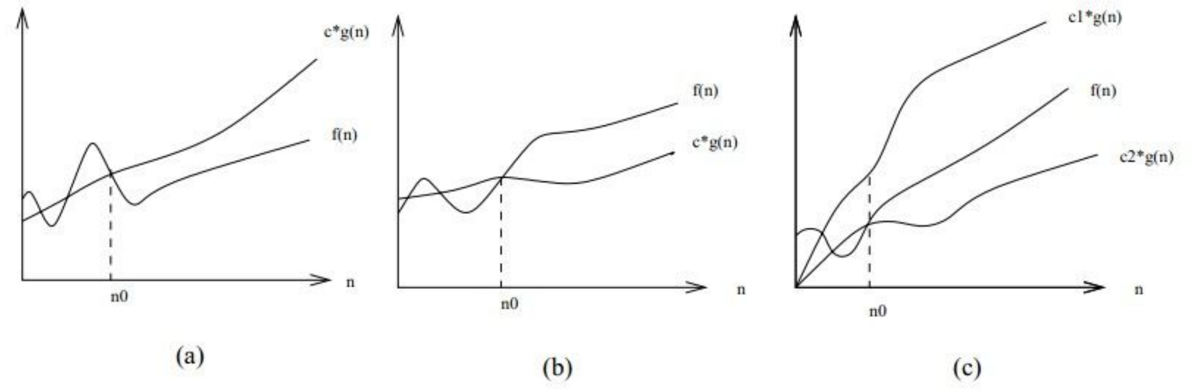


Figure 2.3: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

Complejidad Computacional

Cotas usuales

- **Crecimiento constante**: O(1)
- **Crecimiento logarítmico**: O(log n)
- **Crecimiento lineal**: O(n)
- **Crecimiento cuadrático**: $O(n^2)$
- **Crecimiento polinomial**: $O(n^c)$
- **Crecimiento exponencial**: $O(c^n)$, $c > 1$
- **Crecimiento factorial**: O(n!)

Complejidad Computacional

Clases: PTIME y EXPTIME...

Clase P (PTIME)

- La clase de complejidad P incluye a todos los problemas de decisión que son computables en tiempo polinomial por una MTD.

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \{\mathsf{DTIME}(n^k) \mid k > 0\}$$

Clase P (PTIME)

- La clase de complejidad P incluye a todos los problemas de decisión que son computables en tiempo polinomial por una MTD.

$$P = \bigcup_{k \in \mathbb{N}} \{\mathrm{DTIME}(n^k) \mid k > 0\}$$

- Teóricamente, los problemas en P son todos "*tratables*".

Clase P (PTIME)

- La clase de complejidad P incluye a todos los problemas de decisión que son computables en tiempo polinomial por una MTD.

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \{\mathrm{DTIME}(n^k) \mid k > 0\}$$

- Teóricamente, los problemas en P son todos "*tratables*".

- En la práctica, $O(n^{1000})$ es tan indeseable como $O(2^n)$.

Clase EXP (EXPTIME)

- La clase de complejidad EXP incluye a todos los problemas de decisión que son computables en tiempo exponencial por una MTD. Es decir, aquellos problemas en $O(2^{p(n)})$, con p(n) un polinomio en función de n.

$$\mathrm{EXP} = \bigcup_{k \in \mathbb{N}} \{\mathrm{DTIME}(2^k) \mid k > 1\}$$

Clase EXP (EXPTIME)

- La clase de complejidad EXP incluye a todos los problemas de decisión que son computables en tiempo exponencial por una MTD. Es decir, aquellos problemas en $O(2^{p(n)})$, con p(n) un polinomio en función de n.

$$\mathrm{EXP} = \bigcup_{k \in \mathbb{N}} \{\mathrm{DTIME}(2^k) \mid k > 1\}$$

Que un problema en EXP no significa necesariamente que no pueda resolverse en tiempo polinomial. De hecho, P ⊆ EXP.

Tiempos de MT no-deterministas

Hasta ahora hemos hablado de la complejidad de problemas decidibles por MT deterministas, pero ¿qué pasa si consideramos MT no-deterministas (MTND)?

Tiempos de MT no-deterministas

Hasta ahora hemos hablado de la complejidad de problemas decidibles por MT deterministas, pero ¿qué pasa si consideramos MT no-deterministas (MTND)?

- L es aceptado en tiempo no-determinístico T si existe una MT no-determinista que acepte L en tiempo T.

Tiempos de MT no-deterministas

Hasta ahora hemos hablado de la complejidad de problemas decidibles por MT deterministas, pero ¿qué pasa si consideramos MT no-deterministas (MTND)?

- L es aceptado en tiempo no-determinístico T si existe una MT no-determinista que acepte L en tiempo T.

La clase de todos los lenguajes decidibles en tiempo T por una MTND se denota NTIME(T) (non-deterministic time).

Tiempos de MT no-deterministas

- Un lenguaje es aceptado en tiempo no-determinístico polinomial, si cualquier palabra del lenguaje se acepta con una rama de computación que requiere a lo más un polinomio de n pasos (en el tamaño de la palabra).

Tiempos de MT no-deterministas

- Un lenguaje es aceptado en tiempo no-determinístico polinomial, si cualquier palabra del lenguaje se acepta con una rama de computación que requiere a lo más un polinomio de n pasos (en el tamaño de la palabra).

- En otras palabras, necesitamos que exista una hoja de árbol antes de $T(|w|)$ para que la MTD acepte.

Tiempos de MT no-deterministas

- Un lenguaje es aceptado en tiempo no-determinístico polinomial, si cualquier palabra del lenguaje se acepta con una rama de computación que requiere a lo más un polinomio de n pasos (en el tamaño de la palabra).

- En otras palabras, necesitamos que exista una hoja de árbol antes de T(|w|) para que la MTD acepte.

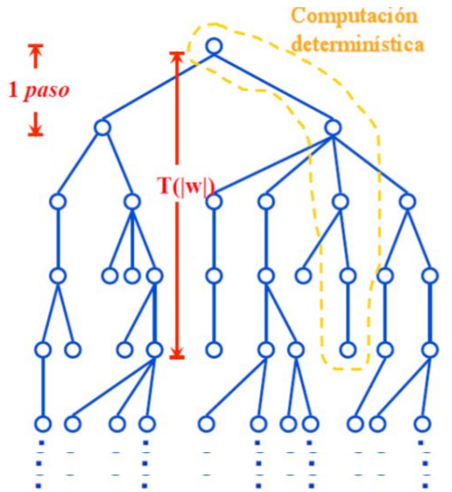
- Podemos ver a una MTD como una MTND con un árbol de ejecución de una rama.

Tiempos de MT no-deterministas

- Un lenguaje es aceptado en tiempo no-determinístico polinomial, si cualquier palabra del lenguaje se acepta con una rama de computación que requiere a lo más un polinomio de n pasos (en el tamaño de la palabra).

- En otras palabras, necesitamos que exista una hoja de árbol antes de T(|w|) para que la MTD acepte.

- Podemos ver a una MTD como una MTND con un árbol de ejecución de una rama.



Computaciones infinitas (no acepta)

Clase NP

- Entendemos como NP (por Non-Deterministic Polinomial) a la clase de lenguajes:

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \{\mathrm{NTIME}(n^k) \mid k > 0\}$$

- La clase NP incluye todos los problemas de decisión computables en tiempo polinomial por una MTND.

- Note que $P \subseteq NP \subseteq EXP$.

Relacionemos ciertas cosas

- Podemos ver a una MTD como una MTND con un árbol de ejecución de una rama.

Por lo anterior,

MTD es subconjunto de MTND

Relacionemos ciertas cosas

- Podemos ver a una MTD como una MTND con un árbol de ejecución de una rama.

Por lo anterior,

MTD es subconjunto de MTND

⬇

L pertenece a DTIME(n^d) => L pertenece a NTIME(n^d)

Relacionemos ciertas cosas

- Podemos ver a una MTD como una MTND con un árbol de ejecución de una rama.

Por lo anterior,

MTD es subconjunto de MTND

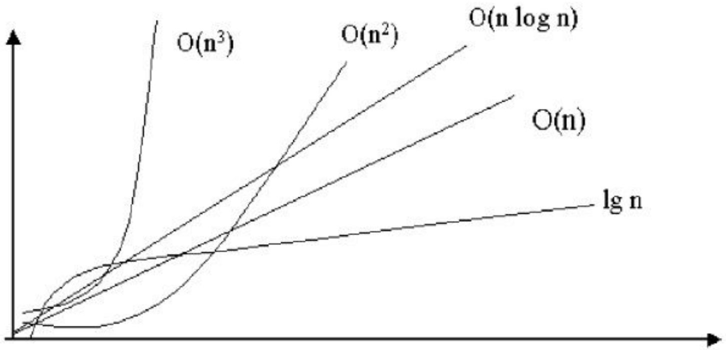
$\Downarrow$

L pertenece a DTIME(n^d) => L pertenece a NTIME(n^d)

$\Downarrow$

La clase P es subconjunto de la clase NP

Complejidad Computacional

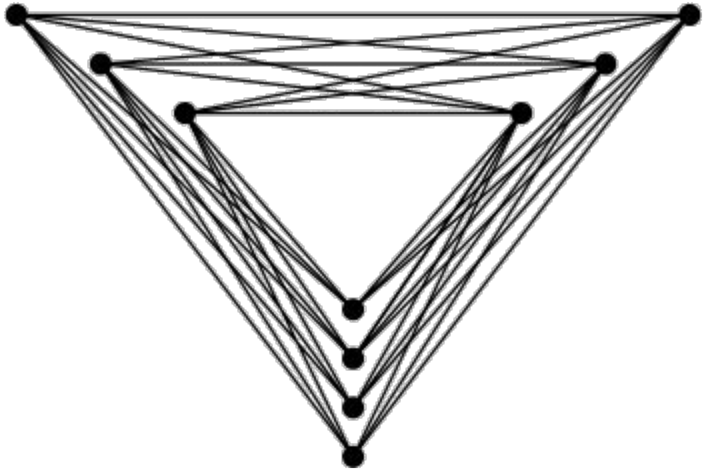


Complejidad / Nro de pasos	10	100	1000
$O(n)$	0,010"	0,100"	1"
$O(n \log_2 n)$	0,033"	0,664"	9,966"
$O(n^2)$	0,1"	10"	16,7 min
$O(n^3)$	1"	16,7 min	11,57 días
$O(2^n)$	1,024"	4 x 10^{13} millones de años	3,4 x 10^{284} millones de años
$O(3^n)$	59,049"	1,63 x 10^{31} millones de años	4,2 x 10^{461} millones de años

Departamento de Ingeniería Informática
Universidad de Santiago de Chile

Teoría de la Computación

Primer semestre 2024



Santiago, 2024